Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Anej Budihna

Algoritmi za zunanje urejanje

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in informatika

Mentor: doc. dr. Jurij Mihelič

Ljubljana 2015

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Problem urejanja zaporedja podatkov v izbranem vrstnem redu je eden izmed osnovnih računskih problemov, ki je vsestransko uporaben na področju računalništva, še posebej algoritmike. Glede na medij hranjenja podatkov ločimo dve vrsti problemov: notranje in zunanje urejanje. Prva vrsta ignorira omejitev velikosti glavnega pomnilnika, druga pa jo upošteva, saj gre za urejanje tako velike količine podatkov, da je vse ne moremo hkrati hraniti v glavnem pomnilniku.

V okviru diplomske naloge preglejte področje zunanjega urejanja. Opišite osnovne in napredne algoritme reševanja problema. Poleg tega preglejte še tehnike pohitritve, kot je predurejanje, in morebitne novejše pristope, ko so predpomnilinško nezavedni algoritmi. Izbrane algoritme učinkovito implementirajte in jih eksperimentalno primerjajte na velikih vhodnih zaporedjih ter smiselno predstavite rezultate testiranja.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Anej Budihna, z vpisno številko **63110188**, sem avtor diplomskega dela z naslovom:

Algoritmi za zunanje urejanje

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Jurija Miheliča,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 16. septembra 2015 Podpis avtorja:

Zahvaljujem se mentorju doc. dr. Juriju Miheliču za strokovno pomoč in nasvete pri izdelavi diplomske naloge. Rad bi se zahvalil tudi družini, ki me je podpirala in spodbujala tekom študija.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
CPU	central processing unit	centralno procesna enota
RAM	random access memory	pomnilnik z naključnim dostopom
DRAM	dynamic random access memory	dinamični pomnilnik z naključnim dostopom
SSD	random access memory	pomnilnik z naključnim dostopom
RPM	rotation per minute	obrati na minuto

Povzetek

V diplomskem delu je predstavljeno področje zunanjega urejanja. V nalogi je opisanih in primerjanih več algoritmov za zunanje urejanje, kako se obnašajo ter katere so njihove prednosti in slabosti. Algoritmi, ki jih primerjamo so navadno večsmerno zlivanje, navadno uravnoteženo zlivanje, naravno uravnoteženo zlivanje, polifazno zlivanje, kaskadno zlivanje, urejanje s porazdeljevanjem, urejanje z lijakom in dva algoritma za predurejanje. Namen diplomske naloge je opisati in predstaviti delovanje teh algoritmov v teoriji in v praksi. Algoritme smo implementirali v programskem jeziku C ter jih med seboj poskusno primerjali na osebnem računalniku z eno zunanjo napravo za shranjevanje podatkov.

Ključne besede: urejanje, podatki, algoritmi.

Abstract

The thesis presents the field of external sorting. In the thesis we describe and compare multiple sorting algorithms for external sorting based on their behavior, their advantages and disadvantages. The algorithms we compare are the straight multiway merge sort, balanced multiway merge sort, natural multiway merge sort, polyphase merge sort, cascade sort, distribution sort, funnel sort and two pre-sorting algorithms. The purpose of the thesis is to describe and present how the algorithms work in theory and in practice. We implemented the algorithms in the C programming language and then experimentally compared them on a personal computer with one external storage device.

Keywords: sorting, data, algorithms.

Poglavje 1

Uvod

Urejanje je proces, kjer se izvaja premeščanje elementov za dosego urejene množice elementov po izbranih kriterijih. Urejanje lahko predstavlja samostojen problem ali pa vmesen korak v večjem procesu. Je eden izmed temeljnih in najpogosteje izvajanih procesov in se izvaja skoraj na vseh možnih področjih kjer se uporabljajo podatkovne baze. Iz tega razloga je urejanje podatkov še vedno zelo relevantna tema in ključna procedura, še posebej na področju procesiranja podatkov. Zato je smiselno, da je bilo opravljenih veliko študij in analiz za razvoj novih in izboljšavo obstoječih algoritmov za urejanje. Pri navadnem urejanju, ki se izvaja znotraj glavnega pomnilnika, se pohitritve dosega z zmanjšanjem procesorske zahtevnosti preko minimizacije števila primerjav. Ti algoritmi se uporabljajo na relativno majhnih količinah podatkov, ki se jih lahko vse na enkrat naloži v glavni pomnilnik. Ko pa se ureja ogromne količine podatkov, ki jih ne moremo vseh hkrati hraniti v glavnem pomnilniku je govora o zunanjem urejanju. Pri zunanjem urejanju se med urejanjem uporablja zunanje pomnilnike, ki lahko hranijo več podatkov a so hkrati veliko počasnejši od notranjega pomnilnika v računalniku.

Splošni postopek pri zunanjem urejanju je ta, da se podatke razdeli na manjše, obvladljive dele, ki se jih na koncu združi v eno urejeno datoteko. Iz tega pristopa izhaja veliko različnih algoritmov, vsak s svojimi prednostmi in slabosti. Glavno ozko grlo, ki ga ti algoritmi skušajo premostiti, predstavlja

prenos podatkov med glavnim in zunanjim pomnilnikom, zato se zmogljivost algoritmov lahko meri glede na število bralnih in pisalnih dostopov.

V prvem poglavju smo najprej predstavili osnove modela zunanjega pomnilnika. Opisali smo notranji in zunanji pomnilnik, zakaj se uporabljata in glavne razlike med njima. Sledi teoretična primerjava algoritmov iz treh različnih paradigm. Pri vsaki smo opisali glavne posebnosti in enega ali več algoritmov, ki smo jih tudi realizirali v programskem jeziku C. V drugem poglavju smo algoritme preizkusili in jih tudi med seboj primerjali.

V dodatku je priloženih nekaj primerov izvorne kode iz samih algoritmov, ki smo jih implementirali.

Poglavje 2

Teoretične osnove

2.1 Pomnilniška hierarhija

Zunanje urejanje je močno odvisno od pomnilniške hierarhije sistema na katerem se izvaja. Običajno se ti algoritmi izvajajo na računalnikih za splošne namene. Iz ekonomskih razlogov imajo takšni računalniki več pomnilniških nivojev, kjer ima vsak nivo drugačne zmogljivosti in cene. Na najnižjem nivoju se nahajajo registri centralne procesne enote, ki predstavljajo najhitrejši a hkrati tudi najdražji pomnilnik. Uporabljajo se izključno kot predpomnilnik za ukaze in podatke, ki jih procesor potrebuje oziroma, ki jih bo potreboval v bližnji prihodnosti. Za notranji glavni pomnilnik se običajno uporablja dinamični pomnilnik z naključnim dostopom. Na najvišjem nivoju pa se nahajajo zunanji pomnilniški mediji, kot so magnetni diski oziroma magnetni trakovi in optični diski, ki omogočajo trajno in poceni shranjevanje podatkov. Največja razlika v hitrosti prenosa nastopi pri komunikaciji med glavnim in zunanjim pomnilnikom [1]. Slika 2.1 prikazuje primer dvonivojske pomnilniške hierarhije.

Naprave za hranjenje podatkov se tipično uvrščajo v primarni ali glavni pomnilnik in sekundarni ali zunanji pomnilnik. Glavni pomnilnik se večinoma nanaša na bralno-pisalni pomnilnik RAM, medtem ko kot zunanji pomnilnik po navadi služijo naprave kot so trdi diski ali magnetni trak.

Vrsta pomnilnika	Dostopni čas
L1 medpomnilnik	0,9 ns
L2 medpomnilnik	2,8 ns
L3 medpomnilnik	12,9 ns
Glavni pomnilnik (DRAM)	120 ns
SSD disk	50-150 μs
Trdi disk	1-10 ms

Tabela 2.1: Tabela s tipičnimi dostopnimi časi za posamezne nivoje. [2]

V diplomski nalogi uporabljamo naslednje parametre, ki so definirani v modelu zunanjega pomnilnika:

- Velikost problema N: število elementov na zunanjem pomnilniku, ki jih je potrebno urediti.

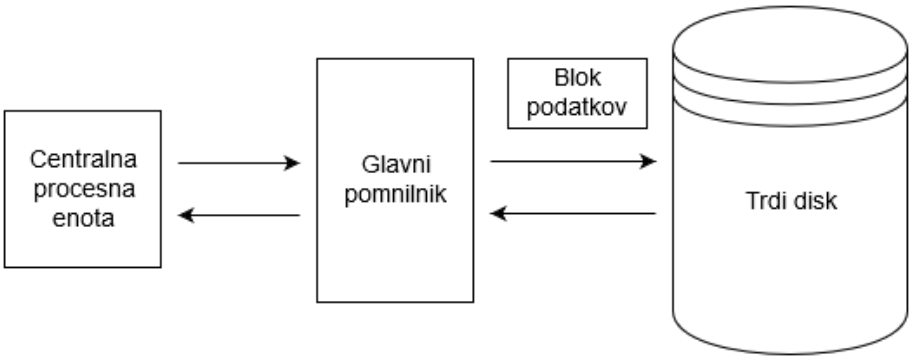
- Velikost glavnega pomnilnika M: število elementov, ki se jih lahko na enkrat shrani v glavni pomnilnik.

- Velikost bloka B: število elementov, ki se jih lahko prenese v posameznem bloku podatkov.

- Število diskov D: število blokov, ki se jih lahko hkrati prenese.

V splošnem velja $1 \leq B \leq M$. V tej diplomski nalogi smo se osredotočili na primere kjer je $D = 1$. Dejanska vrednost teh parametrov je odvisna od implementacije modela. To je naprava na kateri se izvaja algoritem.

V diplomski nalogi pogosto omenjamo trakove. Ti lahko predstavljajo posamezno zunanjo napravo kot sta magnetni trak in magnetni disk, ali pa datoteko na podatkovnem mediju. Datoteke se prav tako lahko nahajajo vsaka na svojem fizičnem pomnilniku ali pa po več skupaj na enem.



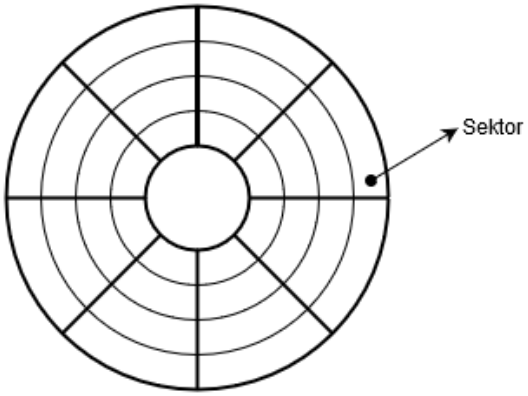
Slika 2.1: Primer pomnilniške hierarhije z dvema nivojema.

2.1.1 Glavni pomnilnik

Pomnilnik z naključnim dostopom se tako poimenuje, ker so dostopni časi do podatkov enaki, ne glede na njivo fizično lokacijo. S časom se velikost in hitrost glavnega pomnilnika v računalnikih povečuje, a ima v primerjavi z zunanjim pomnilnikom dve pomanjkljivosti zaradi katerih ga ne more izpodriniti. Največja pomanjkljivost glavnega pomnilnika, ki ga danes najdemo v računalnikih je ta, da ne omogoča trajnega shranjevanja podatkov. To pomeni, da ko mu odvzamemo napajanje se shranjeni podatki izgubijo. Drugi pomembni faktor je cena. Čeprav danes že obstajajo naprave, ki temeljijo na podobni tehnologiji in omogočajo trajno shranjevanje podatkov je razmerje med ceno in količino podatkov, ki jih lahko hranijo enostavno prevelika za uporabo v podatkovnih centrih. Zato se danes še zmeraj uporabljajo počasnejše zunanje naprave in se bojo zelo verjetno še v prihodnosti.

2.1.2 Trdi disk

Prvi medij za trajno digitalno shranjevanje podatkov je bil magnetni trak. Magnetni trak ima poleg počasnosti dostopov še dodatno omejitev. Zaradi načina delovanja namreč omogoča le serijski dostop do podatkov. Ko je uporabnik v preteklosti želel dostopati do željnega podatka je potrebno lokacijo podatka najprej poiskati. Operacijo iskanja so pohitrili tako, da so podatke



Slika 2.2: Porazdelitev površine plošče v trdem disku.

pred uporabo uredili. Danes se magnetni trakovi še uporabljajo za shranjevanje podatkov a večinoma le za izdelavo arhivskih in varnostnih kopij. V osebnih računalnikih in podatkovnih centrih že več let prevladuje uporaba trdih diskov.

Trdi diski so sestavljeni iz več okroglih kovinskih plošč prevlečenih z magnetno snovjo zloženih eno nad drugo. Plošče so pritrjene na sredinsko vreteno, ki jih med uporabo vrti s stalno hitrostjo. Vsaka površina plošče ima svojo bralno/pisalno glavo, ki preko magnetne sile bere oziroma zapisuje podatke. Površina diska je razdeljena na koncentrične krožnice imenovane sledi, vsaka sled pa je razdeljena na več sektorjev, tako kot je prikazano na sliki 2.2. Sektorji predstavljajo najmanjšo količino podatkov, ki jih disk lahko na enkrat prebere ali zapiše. Več zaporednih sektorjev pa sestavlja blok. Velikost blokov določi operacijski sistem, ko na disku ustvari datotečni sistem. Programer vidi datoteke kot en velik in neprekinjen tok bajtov, v resnici pa je vsaka datoteka na disku zapisana v določenem številu blokov, ki so fizično razpršeni po disku.

Trdi disk ni omejen na serijske prenose in ob vsakem času omogoča dostop vseh pomnilniških lokacij, a ne nujno v enakem dostopnem času. Ko se izvaja branje z diska se mora bralno/pisalna glava premakniti nad ustrezen trak, disk pa se mora zavrteti na ustrezni sektor. Čas, ki se zato porabi se imenuje

iskalni čas in predstavlja glavni del časa pri dostopanju do podatkov. Ko se bralna glava nahaja na ustreznem mestu se podatki relativno hitro prenesejo.

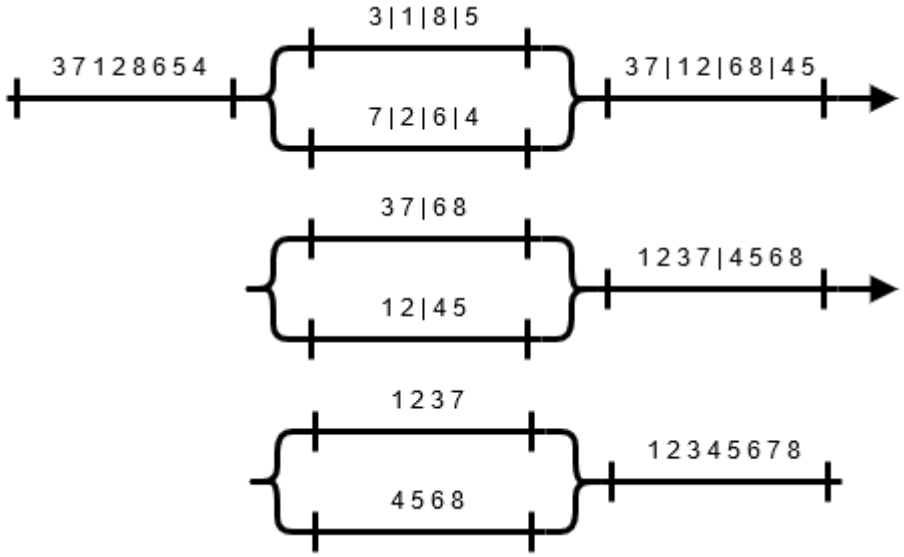
Iz tega razloga je branje in pisanje večjega števila zaporednih sektorjev in blokov časovno ugodnejše od dostopa naključnih lokacij na disku. Zato je smiselno, da se dostope do diska združuje v manjše število večjih prenosov. Ta postopek se imenuje predpomnjenje in izmed ključnih inženirskih trikov, ki se uporablja za pospeševanje algoritmov, ki komunicirajo z zunanjimi pomnilniki.

2.2 Osnovni algoritmi zlivanja

Urejanje z zlivanjem je eden izmed najstarejših pristopov in tudi eden izmed najbolj pogosto uporabljenih pri urejanju podatkov. Osnovni algoritmi so nastali že v času, ko se je za shranjevanje podatkov uporabljalo magnetne trakove. Algoritmi podatke najprej razdelijo na več manjših urejenih kosov in jih nato v več korakih združi v eno samo urejeno zaporedje. Skozi čas se je pojavilo več takšnih algoritmov, z različnimi metodami porazdeljevanja in zlivanja [3]. V tem poglavju opisujemo nekatere izmed glavnih algoritmov, ki temeljijo na tem pristopu.

2.2.1 Navadno neuravnoteženo zlivanje

Navadno zlivanje je eden izmed prvih in osnovnejših algoritmov, ki služi kot temelj za prihodnje algoritme. Osnovni algoritem uporablja tri trakove podatkov. Na vhodnem traku se nahajajo elementi, ki jih želimo urediti, uporabljena pa sta še dva prazna delovna trakova. V prvem koraku se iz vhoda zaporedno bere urejena podzaporedja in se jih izmenično zapisuje na delovna trakova tako, da so kar se da enakomerno razporejena. Nato se podzaporedja iz obeh izhodnih trakov prebira in zliva nazaj na vhodni trak. Zlivanje v tem primeru pomeni, da se dve urejeni zaporedji dolžine k združi v eno urejeno zaporedje dolžine $2k$. Pri prvem porazdeljevanju so zaporedja sestavljena iz samo enega elementa. Ta postopek porazdeljevanja in zlivanja



Slika 2.3: Primer neuravnoteženega navadanega zlivanja z dvema trakovoma. Črte med števili ločujejo urejena zaporedja.

se ponavlja in tako zliva čedalje večja zaporedja dokler ne ostane samo eno urejeno zaporedje. Primer navadnega neuravnoteženega zlivanja je prikazan na sliki 2.3

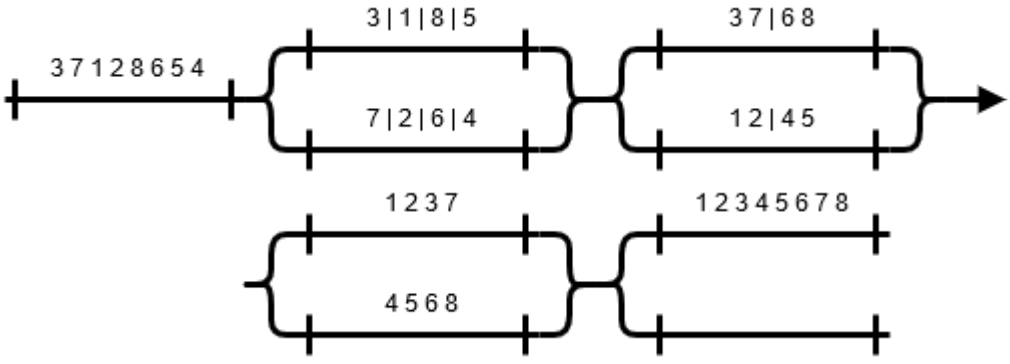
Prednost tega algoritma je, da porabi samo tri trakove in ne potrebuje velikega notranjega pomnilnika. Algoritem je mogoče pohitriti z uporabo dodatnih delovnih trakov. Na ta način se zaporedja hitreje povečujejo, kar zmanjša potrebno število faz porazdeljevanja in zlivanja ter posledično tudi število pomnilniških dostopov. Takšno urejanje se imenuje večsmerno navadno zlivanje.

Zaradi preprostosti algoritma je mogoče čas izvajanja zelo enostavno oceniti. Če algoritem poženemo nad N podatkovnimi zapisi, moramo za ureditev elementov izvajati $log_2(N)$ faz porazdeljevanja in zlivanja. Ker v vsaki fazi podatke enkrat preberemo in zapišemo je čas, ki ga navadno zlivanje porabi enak $\Theta(Nlog_2(N))$. Če vpeljemo še uporabo n trakov postane zahtevnost algoritma enaka $\Theta(Nlog_n(N))$ [4].

Implementacija. Implementacija algoritma je razmeroma enostavna. Uporabili smo strukturo v kateri hranimo podatke o traku kot je pot do datoteke, ki jo uporabljamo in število elementov zaporedja, ki jih moramo zliti. Ker želimo pri urejanju ohraniti izvorno datoteko v tem primeru uporabljamo dodaten trak za zlivanje. V prvem klicu kot vhod za porazdeljevanje uporabimo izvorno datoteko, v vseh ostalih pa uporabljamo dodatni trak. Po vsakem zlivanju je potrebno elemente ponovno porazdeliti. Število trakov, ki se uporablja med izvajanjem je mogoče definirati ob zagonu. Algoritem A.1 prikazuje kodo za zlivanje elementov iz več vhodnih trakov na en izhodni.

2.2.2 Navadno uravnoteženo zlivanje

Navadno uravnoteženo zlivanje je še ena osnovna izboljšava prejšnjega algoritma. Faza porazdeljevanja je odvečna saj ne vpliva na urejenost podatkov. Z manjšo izboljšavo je mogoče fazo porazdeljevanja združiti s fazo zlivanja. Za ta algoritem sta potrebna vsaj dva vhodna in dva izhodna trakova. V prvem koraku algoritma ostaja porazdeljevanje elementov iz vira na delovne trakove. V vseh prihodnjih korakih pa se zaporedja že med zlivanjem izmenično izpisujejo na pomožna izhodna trakova. Po vsaki fazi zlivanja se vloge trakov zamenjajo. Primer uravnoteženega zlivanja je prikazan na sliki 2.4.



Slika 2.4: Primer navadanega uravnoteženega zlivanja z dvema takovoma. Črte med števili ločujejo urejena zaporedja.

Implementacija. Pri implementaciji smo nadgradili kodo prejšnjega algoritma. V prvem koraku izvajamo porazdelitev elementov iz vhodne datoteke, v vseh naslednjih pa samo še zlivanje. Algoritem A.2 prikazuje glavni del spremenjene kode izboljšanega algoritma za zlivanje. Vsakič, ko končamo z zlivanjem enega zaporedja se premaknemo na naslednji izhodni trak. Ko za zlivanje uporabimo samo en trak vemo, da so urejeni podatki v enem kosu in lahko zato končamo z zlivanjem.

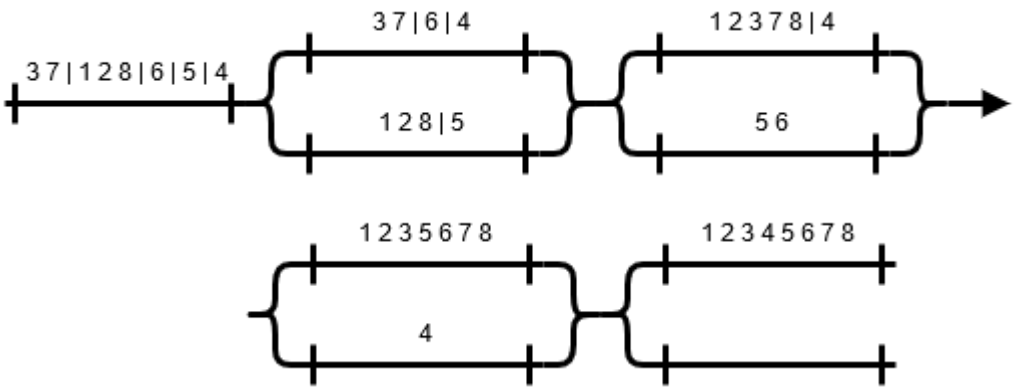
2.2.3 Naravno zlivanje

Pri navadnem zlivanju se zliva zaporedja dogovorjenih dolžin in se vedno začne z zaporedji s samo enim elementom. Toda pogosto so podatki na vhodu že delno urejeni, kar se lahko v algoritmu izkoristi. Naravno zlivanje namesto zaporedij dogovorjenih dolžin uporablja čete. Četa je vsako urejeno zaporedje, ki se ga ne da razširiti ne da bi izgubilo svojo urejenost. Če je a_i prvi element čete in a_j zadnji element čete velja:

$$a_{i-1} \leq a_i \leq ... \leq a_j \leq a_{j+1}$$

Bolj kot so vhodni podatki urejeni boljša je učinkovitost tega algoritma. V primeru, da so podatki naključno razporejeni pa pospešitev postane majhna oziroma skoraj zanemarljiva. Uporaba čet hkrati pomeni, da je potrebno pri zlivanju sproti preverjati ali prebrani element še sodi v četo. Zato se v pomnilniku hrani zadnji prebrani element, ki se ga nato uporabi za primerjavo z novim prebranim elementom. Posledično porabi algoritem nekaj več procesorskega časa, a je ta še vedno zanemarljiv v primerjavi s časom, ki se ga porabi za dostopanje do podatkov. Primer naravnega zlivanja je prikazan na sliki 2.5.

Implementacija. Algoritem je v primerjavi z navadnim uravnoteženim zlivanjem večinoma nespremenjen. Pri implementaciji algoritma smo prvo porazdeljevanje in zlivanje spremenili tako, da za ugotavljanje konca čete izvaja primerjavo med novo in prejšnjo prebrano vrednostjo. Če je nova vrednost



Slika 2.5: Primer naravnega zlivanja z dvema takovoma. Črte med števili ločujejo urejena zaporedja.

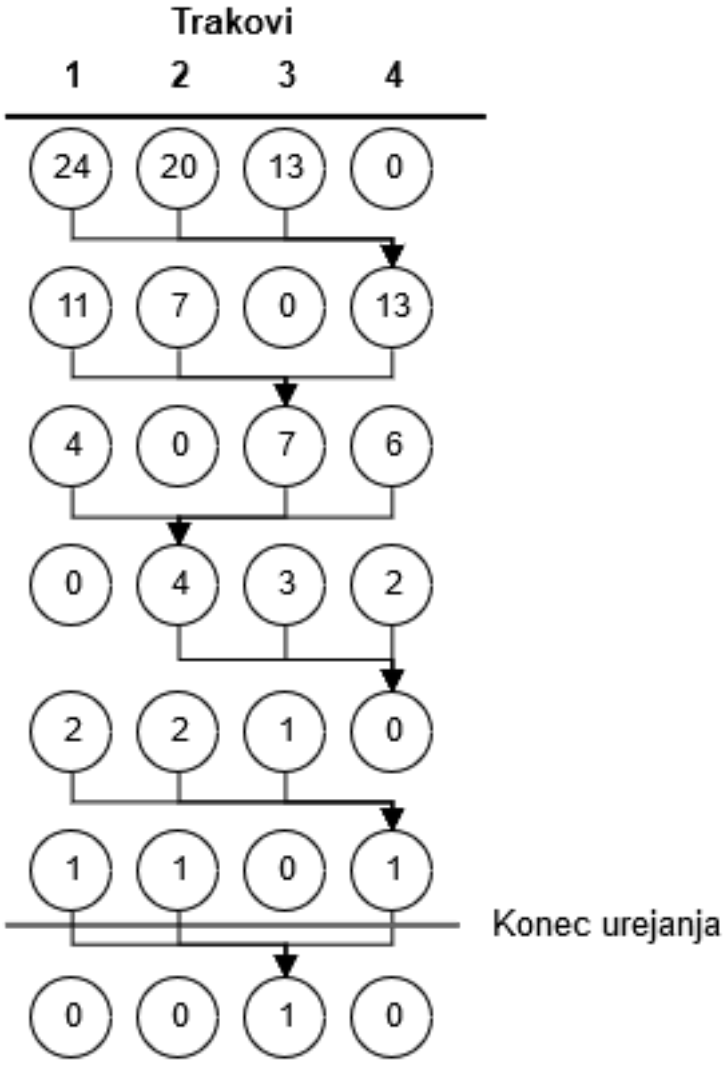
manjša od prejšnje se četa zaključi. Spremembe, ki smo jih uvedli pri fazi zlivanja so prikazane v algoritmu A.3.

2.3 Napredni algoritmi zlivanja

2.3.1 Polifazno zlivanje

Pri prejšnjih algoritmih, ki uporabljajo več izhodov je vedno na enkrat aktiven samo en izhodni trak medtem, ko ostali mirujejo. Polifazno zlivanje združuje prejšnje izboljšave in hkrati skuša doseči boljšo izkoriščenost trakov tako, da se za izhod vedno uporablja samo en trak, vse preostale pa za branje. Pri vsaki fazi zlivanja se izprazni en trak, ki nato v naslednji fazi prevzame vlogo izhodnega traku. Za to, da se v vsaki fazi izprazni natanko en vhodni trak in da se algoritem lahko do konca pravilno izvede morajo biti v vsaki fazi čete porazdeljene na točno določen način.

Za ugotavljanje števila čet, ki jih mora določen trak vsebovati se uporablja zaporedje, ki izvira iz Fibonaccijevega zaporedja števil. Pri Fibonaccijevem zaporedju je vsak naslednji člen enak vsoti prejšnjih dveh členov zaporedja.



Slika 2.6: Primer zlivanja pri polifaznem urejanju s tremi vhodnimi in enim izhodnim trakom.

n	a_1^n	a_2^n	a_3^n	a_4^n	$\sum a^n$
0	1	0	0	0	1
1	1	1	1	1	4
2	2	2	2	1	7
3	4	4	3	2	13
4	8	7	6	4	25
5	15	14	12	8	49
6	29	27	23	15	94
$n+1$	$a_1^n + a_2^n$	$a_1^n + a_3^n$	$a_1^n + a_4^n$	a_1^n	

Tabela 2.2: Primer optimalne porazdelive za polifazno zlivanje s štirimi vhodnimi in enim izhodnim trakom.

Torej, če i predstavlja indeks števila v zaporedju velja:

$$f_i = f_{i-1} + f_{i-2}$$

To je Fibonaccijevo zaporedje drugega reda. Red nam pove število predhodnih členov, ki jih moramo sešteti, da dobimo naslednje število v zaporedju. Če gledamo prvi stolpec v tabeli 2.2 vidimo, da se navpično navzdol vrstijo Fibonaccijeva števila reda štiri. Med števili v tabeli obstaja še ena relacija, ki nam poenostavi računanje števil za določen nivo. In sicer jih je mogoče izračunati s pomočjo naslednje formule:

$$a_i^n = a_1^{n-1} + a_{i+1}^{n-1}$$

V enačbi predstavlja i indeks števila v vrstici oziroma zaporedno številko traku, n pa nivo razporeditve. Pri računanju začnemo z določitvijo ničtega nivoja. Ta nivo predstavlja rezultat zadnje faze zlivanja, kjer se na enem izhodnem traku nahaja ena urejena četa. Z uporabo prej omenjene enačbe lahko tako iz tega nivoja izpeljemo porazdelitve za ostale nivoje [5].

Za to, da lahko določimo ustrezen nivo porazdelitve moramo poznati število uporabljenih trakov in število čet, ki jih urejamo. Da se izognemo štetju čet pred izvajanjem algoritma lahko nivo razporeditve računamo kar

med samim porazdeljevanjem. Začnemo s porazdelitvijo na nivoju ena, kjer ima vsak trak prostora za eno četo in začnemo s porazdeljevanjem. Če nam po končani fazi zmanjka prostora za čete in vhodni trak še ni prazen povečamo nivo porazdelitve in nadaljujemo s porazdeljevanjem. Ta korak ponavljamo dokler ne izpraznimo vhodnega traku. V primeru, da nam v zadnji fazi porazdeljevanja čet zadnji fazi zmanjka čet in nam ne uspe zapolniti vseh trakov manjkajoče čete zapišemo v obliki navideznih čet. Navidezne čete uporabimo samo zato, da se algoritem lahko pravilno izide. Hranimo jih tako, da za vsaki trak vpeljemo svoj števec navideznih čet. Pri zadnji fazi porazdelitve nato nastavimo števec na število čet, ki nam manjka za dopolnitev zaporedja [6].

Navidezne čete vpeljejo posebnosti tudi v algoritem za zlivanje trakov. Ko algoritem naleti na trak z navidezno četo ne prebere resnične čete iz traku, temveč samo zmanjša števec navideznih čet za dani trak. V primeru, da imajo vsi vhodni trakovi navidezne čete, se pri zlivanju zmanjša števec navideznih čet vhodnih trakov in poveča števec navideznih čet izhodnega traku.

Algoritem je mogoče pohitriti tako, da se navidezne čete enakomerno porazdeli med trakove. Na ta način se navidezne čete zlijejo med seboj, kar nam prihrani veliko odvečnega zapisovanja in omogoča, da velikost resničnih čet med zlivanjem hitreje narašča. To najlažje dosežemo tako, da pri porazdeljevanju resnične čete vedno naprej dodelimo tistemu traku, ki ima v danem trenutku največje število prostih mest.

Implementacija. Pri implementaciji smo strukturo za hranjenje podatkov traku dodatno spremenli. V strukturi hranimo število čet, ki bi jih datoteka morala hraniti in število navideznih čet, ki smo jih uporabili za dopolnitev zaporedja. Za porazdeljevanje posamezne čete vedno izberemo trak katerumu manjka največ čet do zapolnitve zaporedja. Ker uporabljamo navidezne čete moramo med zlivanjem izvajati dodatna preverjanja zato, da se čete pravilno zlivajo in prenašajo v naslednje nivoje. To v kombinaciji s posebnim

zaporedjem zlivanja trakov pomeni, da je alogritem za zlivanje A.4 nekoliko zahtevnejši.

2.3.2 Kaskadno zlivanje

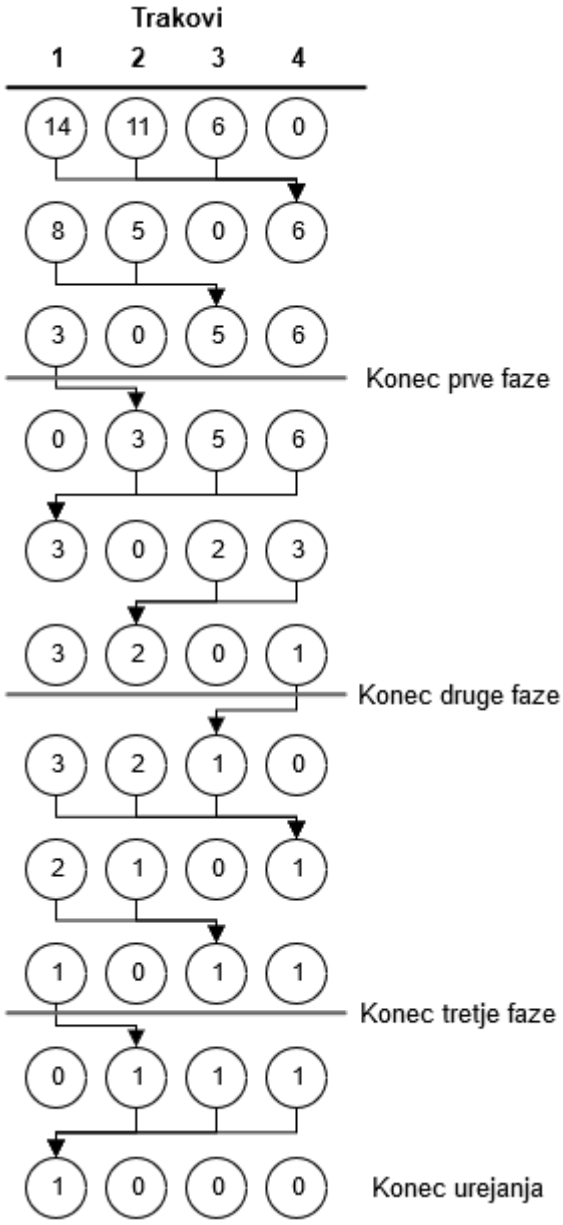
Kaskadno zlivanje, podobno kot polifazno zahteva, da so čete razporejene v določenem zaporedju, le da je zaporedje nekoliko drugačno. Za primer porazdelitev in splošno pravilo glej tabelo 2.3.

n	a_1^n	a_2^n	a_3^n	a_4^n	$\sum a^n$
0	1	0	0	0	1
1	1	1	1	1	4
2	4	3	2	1	10
3	10	9	7	4	30
4	30	26	19	10	85
5	85	75	56	30	246
6	246	216	160	85	707
$n+1$	$a_1^n + a_2^n + a_3^n + a_4^n$	$a_1^n + a_2^n + a_3^n$	$a_1^n + a_2^n$	a_1^n	

Tabela 2.3: Primer optimalne porazdelitve za kaskadno zlivanje s štirimi vhodnimi in enim izhodnim trakom.

Medtem, ko polifazno zlivanje na vsakem nivoju združi le delež vseh čet, pri kaskadnem urejanju vsak nivo porazdelitve predstavlja zlivanje vseh čet. V primeru s tremi vhodnimi trakovi in enim izhodnim trakom algoritem začne z zlivanje prvih treh trakov na četrtega, ki je prazen in s tem izprazni tretji trak. Nato zlije prva dva trakova na tretjega in na koncu še prepiše podatke iz prvega traku na drugega. Ta postopek se nadaljuje dokler niso vsi podatki urejeni na enem traku. Za boljše razumevanje izvajanja algoritma glej sliko 2.7.

Očitno je, da je kopiranje podatkov, ki se izvede v zadnjem koraku zlivanja odvečno in se mu je mogoče izogniti z enostavno zamenjavo zadnjih dveh traku. Čeprav se izkaže, da se kopira le manjši del elementov in kopiranje



Slika 2.7: Primer zlivanja pri kaskadnem urejanju s tremi vhodnimi in enim izhodnim trakom.

zasede le majhen procent celotnega časa postopka urejanja. Metoda zlivanja se ne zdi optimalna, še posebej v primerjavi z polifaznim zlivanjem, ki pri vsakem zlivanju izkoristi vse trakove. Toda izkaže se da je kaskadno zlivanje asimptotično boljše, če se uporabi šest ali več trakov. Glavni razlog za to je, da pri isti količini podatkov zahteva manjši nivo porazdelitve. Zato se v splošnem algoritem hitreje izvede od polifaznega [7].

Porazdeljevanje elementov je zelo podobno porazdeljevanju pri polifaznem urejanju, saj lahko na podoben način porazdelitev računamo med porazdeljevanjem in za dopolnitev porazdelitve uporabljamo navidezne čete. Za računaje število potrebnih elementov na vsakem traku uporabimo naslednjo formulo:

$$a_i^n = \sum_{j=1}^{k-(i-1)} a_j^{n-1},$$

kjer je i indeks traku, n nivo porazdelitve in k skupno število trakov.

Implementacija. Algoritem smo izpeljali iz polifaznega zlivanja, zato je upravljanje z navideznimi četami in porazdeljevanje čet skoraj enako. Kot smo že omenili, glavne posebnosti nastopajo pri zlivanju. Med zlivanjem zmanjšujemo število trakov iz katerih beremo podatke in na koncu vsake faze obrnemo vrstni red trakov. Spremenjeni algoritem za zlivanje je vsebovan v dodatkih A.5.

2.4 Predurejanje

Do sedaj smo večinoma opisovali izboljšave pri fazi zlivanja, obstajajo pa tudi izboljšave, ki jih lahko izvedemo pri prvem porazdeljevanju. Cilj predurejanja je, da z uporabo glavnega pomnilnika predhodno uredi podatke in ustvari kar se da dolga urejena zaporedja ali čete. Na ta način zmanjša število potrebnih faz zlivanj glavnega algoritma in posledično močno pohitri izvajanje.

2.4.1 Uporaba notranjega urejanja

Najenostavnejši pristop pri predurejanju je ta, da se vhodno datoteko uredi po kosih fiksne dolžine. V vsakem koraku se prebere čim večjo količino podatkov v glavni pomnilnik in se jih nato uredi z algoritmom, ki urejanje opravi v času $\Theta(n \log(n))$.

Urejeni kos podatkov se nato razporedi na ustrezni trak kot eno urejeno zaporedje. Prednost tega algoritma je, da je enostaven za implementacijo in se lahko uporablja v kombinaciji z algoritmi, ki zahtevajo fiksne dolžine čet.

2.4.2 Predurejanje s kopico

Predurejanje s kopico (angl. replacement selection) je naprednejši algoritem, ki bolje izkorišča velikost glavnega pomnilnika. V povprečju ustvari čete, ki so dvakrat večje od števila elementov, ki se jih lahko shrani v uporabljeni glavni pomnilnik [8]. Metoda temelji na uporabi kopice, ki je definirana, kot binarno drevo, v katerem ima vsako vozlišče manjšo vrednost on njegovih naslednikov. Algoritem deluje na sledeč način :

1. Napolni glavni pomnilnik s podatki iz vhodne datoteke.
2. Zgradi kopico.
3. Nato dokler ne zmanjka podatkov:
   (a) Zapiši najmanjšo vrednost iz kopice v izhodno datoteko. Vrednost se nahaja v korenu drevesa in na prvem mestu.
   (b) Preberi novo vrednost iz vhodne datoteke.
      i. Če je vrednost večja od prejšnje zapisane vrednosti jo postavi v koren kopice.
      ii. Če je vrednost manjša jo postavi zadnjo vrednost v kopici na prvo mesto in prebrano vrednost na konec kopice. Nato zmanjšaj velikost kopice.
   (c) Kopico pogrezaj.

Med procesom se pomnilnik razdeli na dva dela. Prvi del je kopica, drugi del pa predstavljajo elementi, ki se bodo uporabili pri ustvarjanju naslednje čete. S tem ko se kopica manjša ustvarja prostor za elemente, ki ne sodijo v četo. Ko se kopica do konca izprazni se četa zaključi. Takrat algoritem vzame prejšnje odvečne elemente iz njih zgradi novo kopico. Algoritem se izvaja dokler ne zmanjka vhodnih elementov. Primer proteka porazdeljevanja je prikan na sliki 2.8.

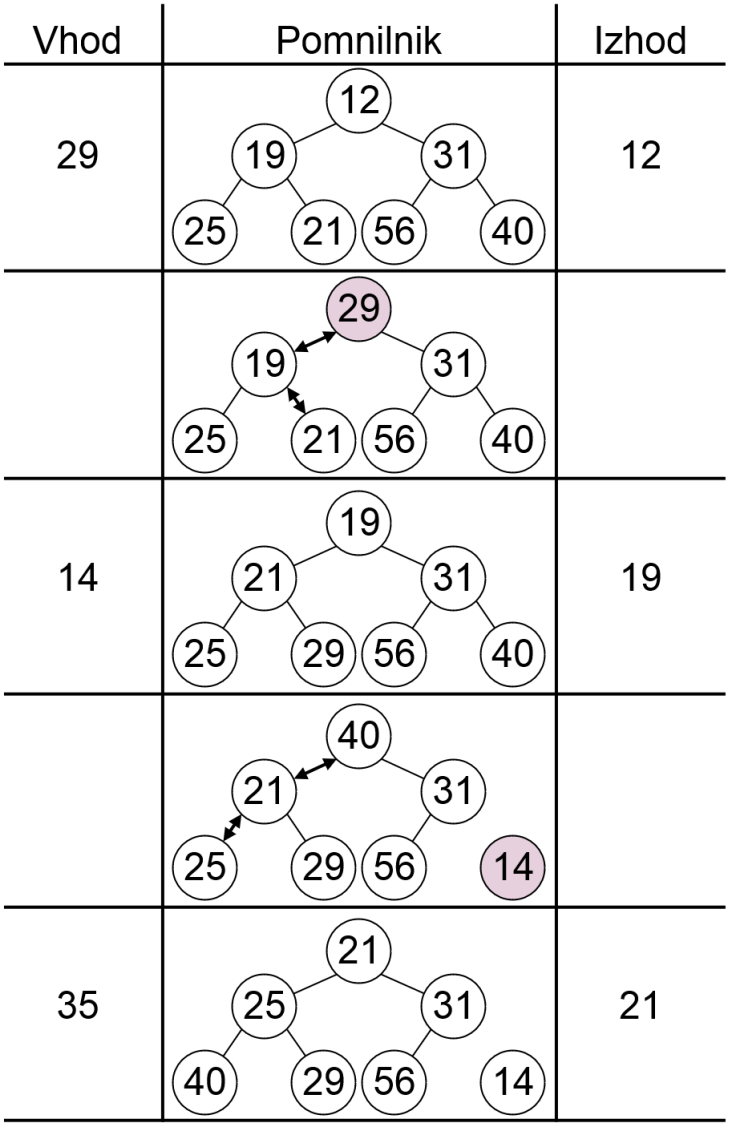
V najboljšem primeru, kjer je večji del datoteke že urejen, se lahko na ta način uredi celotno datoteko. V najslabšem primeru, kjer je datoteka urejena v obratnem vrstnem redu, pa se ustvarijo čete z največjo dolžino M. Ker so podatki v podatkovnih bazah po navadi delno že urejeni, se v praksi izkaže kot zelo dober algoritem. Urejena zaporedja, ki jih algoritem ustvari so različnih velikosti, zato se ga lahko uporablja samo skupaj z algoritmi za zunanje urejanje, ki pri urejanju uporabljajo čete.

Implementacija. Pri implementaciji smo kopico shranjevali tabelo. Uporabljamo dve spremenljivki za beleženje velikosti kopice in števila odvečnih elementov. Ko nam zmanjka elementov na vhodu preostale elemente uredimo z algoritmom za hitro urejanje in jih izpišemo na izhod. Algoritem A.6 prikazuje kos kode, ki upravlja s kopico.

2.5 Urejanje s porazdeljevanjem

Urejanje s porazdeljevanjem je rekurziven proces, kjer se uporablja množico $S-1$ selekcijskih elementov s katerimi se porazdeli vseh N elementov v S neodvisnih pod-datotek. Vsaka pod-datoteka vsebuje elemente, ki so večji od spodnje mejne vrednosti in manjši ali enaki zgornji mejni vrednosti.

Za primer vzamemo množico elementov za porazdeljevanje $e_1, e_2, ..., e_{S-1}$ in poddatoteko i, kjer za i velja $1 \leq i \leq S$. V tem primeru mora poddatoteka vsebovati vse elemente na intervalu $[e_{i-1}, e_{i+1})$, kjer je $e_0 = -\infty$ in $e_S = \infty$.



Slika 2.8: Primer predurejanja s kopico. Po gradnji kopice se vrednost v korenu postavi na izhod, zamenja pa jo vrednost na vhodu. Nato se zvede pogrezanje kopice, ki v koren premakne naslednjo najmanjšo vrednost. Ker je vhodna vrednost 14 premajhna in ne sodi v četo se jo postavi na konec tabele, prejšnjo vrednost pa se postavi v koren.

Pomembna lastnost, ki jo je treba izpolniti pri porazdeljevanju je ta, da vsi elementi v eni pod-datoteki nastopajo pred vrednostmi iz naslednje pod datoteke. Rekurzivno porazdeljevanje se izvaja dokler niso vse poddatoteke dovolj majhne, da se jih lahko v celoti naloži v glavni pomnilnik in uredi, torej dokler ni število zapisov v posamezni datoteki manjši ali enak M. Na ta način se lahko podatke v zadnjem koraku uredi tako, da se v glavnem pomnilniku uredi vsako posamezno poddatoteko, ter da se jih na koncu spoji v eno urejeno celoto.

Prikaz urejanja s porazdeljevanjem. Datoteka na levi strani lahko predstavlja vhodno datoteko ali pa eno izmed poddatotek v eni izmed nižjih stopenj rekurzije.

2.5.1 Izbiranje elementov za porazdeljevanje

Na vsakem nivoju se izvede porazdelitev izbrane datoteke na S pod-datotek. Število S se določi glede na količino dostopnega pomnilnika. Če vsaki datoteki dodelimo B enot medpomnilnika, je največja količina poddatotek, ki jih lahko ustvarimo enaka $O(M/B)$. Ker v zadnji fazi porazdeljevanja ustvarimo poddatoteke velikosti največ M lahko S dodatno omejimo na $O(N/M)$. Iz teh dveh omejitev lahko izpeljemo, da moramo v vsakem koraku porazdeljevanja iz datoteke izbrati $\Theta\left(min\left\{M/B, N/M\right\}\right)$ elementov za porazdeljevanje.

Za iskanje porazdelitvenih elementov obstaja več metod. V glavnem se delijo na deterministične in verjetnostne metode. V primeru da so elementi naključno razporejeni se verjetnostna metoda izkaže kot zelo učinkovita. Dodatna prednost je ta, da je enostavna za implementacijo. V primeru, da se datoteko porazdeljuje na S delov, se iz naključnega dela datoteke prebere dS elementov. Te elemente se nato uredi in izbere vsaki d-ti element.

Implementacija. Na začetku algorima preštejemo število elementov v vhodni datoteki zato, da izvemo na koliko poddatotek jo moramo razdeliti. Med izvajanjem rekurzivne porazdelitve hranimo velikosti generiranih poddatotek v namenski podakovni strukturi. Porazdeljevanje izvajamo dokler ni podda-

toka dovolj majhna, da jo lahko v celoti hranimo v glavnem pomnilniku. Količino pomnilnika, ki jo algoritem uporabi se nastavi ob začetku izvajanja preko argumenta v ukazni vrstici. V primeru da je vhodna datoteka dovolj majhna se porazdeljevanje preskoči. Po vsaki končani porazdelitvi, porazdeljeno poddatotko izbrišemo zato, da zmanjšamo količino prostora, ki ga algoritem uporablja med urejanjem.

Za izbiranje množice elementov za porazdeljevanje smo uporabili verjetnostno metodo. Na začetku algoritma se določi vrednosti parametrov. Najprej se izvede naključen zamik v datoteki in nato prebere dS elementov, kjer je d naključno generirano število, S pa število poddatotek, ki jih želimo ustvariti. Tabelo elementov hitro uredimo in izberemo porazdeljevalne elemente tako, da izberemo vsaki d-ti element. Algoritem za izbiranje elementov za porazdeljevanje je vsebovan v dodatku A.7.

2.6 Pomnilniško nezavedni algoritmi

Pomnilniško zavedni (angl. cache aware) in pomnilniško nezavedni (angl. cache oblivious) algoritmi predstavljajo dva različna pristopa pri zmanjševanju časa izvajanja zunanjega urejanja.

Pomnilniško zavedni algoritmi so osnovani na parametrih, ki se jih lahko natančno prireja in optimizira glede na sistem na katerem se izvajajo. Ti algoritmi imajo predhodno vedenje o velikosti glavnega pomnilnika, pomnilniških blokov in drugih parametrov. V praksi so oblikovani za delovanje na dveh nivojih pomnilniške hierarhije, zato ti algoritmi ne delujejo optimalno na sistemih večjim številom nivojev. Glavni dve pomanjkljivosti teh algoritmov so, da lahko postanejo zelo zapleteni z uporabo velikega števila parametrov in da so odvisni od strojne opreme.

Pomnilniško nezavedni algoritmi pa se uporabi parametrov izogibajo in nimajo podatka o velikosti glavnega pomnilnika. Zato so ti algoritmi neodvisni od strojne opreme in posledično zelo prenosljivi. Dodatna prednost je ta, da če se algoritem optimalno izvaja v pomnilniški hierarhiji z dvema

nivojema se optimalno izvaja tudi na vsakem nivoju katere koli večnivojske pomnilniške hierarhije.

Tako kot številni drugi hitri algoritmi za urejanje podatkov pomnilniško nezavedni algoritmi temeljijo na pristopu deli in vladaj. Pri tem pristopu se glavni problem razdeli na več neodvisnih in bolj obvladljivih problemov. To je ena izmed klasičnih tehnik, ki se pogosto uporablja v razvoju algoritmov. V primeru pomnilniško nezavednih algoritmov pomaga dosegati konstantno in optimalno število pomnilniških prenosov [9].

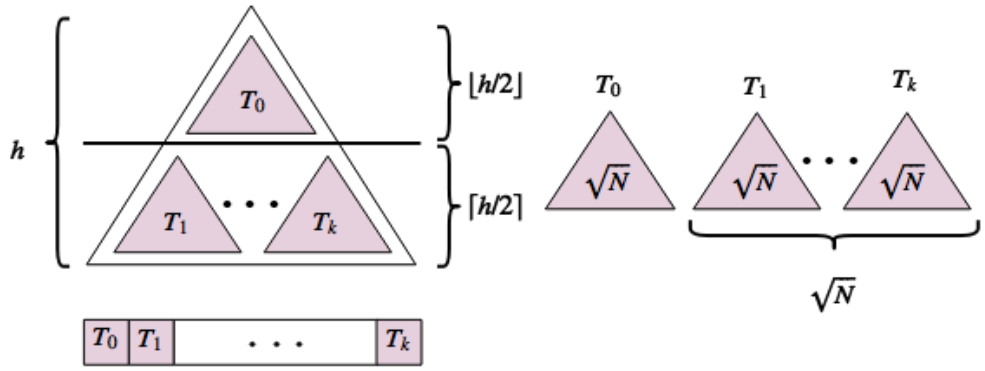
2.6.1 Van Emde Boas raporeditev

Binarno drevo je eno izmed najbolj osnovnih podatkovnih struktur, ki se pogosto uporablja za indeksiranje in iskanje podatkov [10]. Harald Prokop je z uporabo van Emde Boas razporeditve pokazal, kako binarno drevo shraniti v pomnilnik tako, da lahko pridobimo statično in pomnilniško nezavedno binarno drevo [11].

Vzemimo za primer popolno binarno drevo (angl. perfect binary tree). Drevo je polno oziroma popolnoma izravnano, če velja, da so vsi listi na istem nivoju in imajo vsa notranja vozlišča natanko dva naslednika. Iz tega sledi, da ima drevo višino h, skupaj $2^{h+1}-1$ vozlišč in $N = 2^h$ listov. Drevo navidezno razdelimo na dva dela na sredinskem nivoju povezav in tako dobimo eno vrhnje in $\sqrt{N}$ spodnjih poddreves, vsako s približno $\sqrt{N}$ listi. V primeru, da višina drevesa ni potenca števila dva, se drevo razdeli tako, da imajo spodnja poddrevesa višino $\lceil h/2 \rceil$, zgornje poddrevo pa višino $h - \lceil h/2 \rceil$. Slika 2.9 prikazuje porazdelitev za splošni primer.

Najprej se rekurzivno shrani vrhnje poddrevo in nato še spodnja poddrevesa. Toda v praksi vrstni red shranjevanja pravzaprav ni pomemben. Pomembno je le, da je vsako rekurzivno poddrevo shranjeno v enem pomnilniškem segmentu in da so ti segmenti v pomnilniku shranjeni tesno skupaj brez vmesnih praznin [9].

Z uporabo van Emde Boas razporeditve torej dosežemo, da so elementi, ki jih bomo zelo verjetno skupaj potrebovali v istem ali vsaj bližnjem bloku



Slika 2.9: Splošni primer van Emde Boas porazdeliteve za binarno drevo višine h z N listi.

podatkov. Zato ob prenosu enega bloka podatkov iz pomnilnika dobimo več uporabnih podatkov in tako zmanjšamo potrebno število prenosov.

Uporaba te razporeditve v praksi prinese približno 10% pohitritev v primerjavi s klasičnimi pomnilniškimi razporeditvami binarnih dreves [12].

2.6.2 Urejanje z uporabo lijaka

Prvi predstavljeni pomnilniško nezavedni algoritem je bilo urejanje z uporabo lijaka (angl. funnelsort). Algoritem temelji na podatkovni strukturi lijak (angl. funnel) po kateri je tudi poimenovan [11].

Vsaki K-lijak ima K vhodov, ki jih uporabi zato, da ob vsakem klicu zlije K^3 elementov iz K urejenih vhodnih tokov. V bistvu je K-lijak popolno binarno drevo s K-listi, ki je v pomnilniku shranjeno po razporeditvi van Emde Boas. Zato je zaradi rekurzivne definicije vsako poddrevo, ki nastopa v K-lijaku tudi $\sqrt{K}$-lijak. Lijak poleg vozlišč shranjuje tudi medpomnilnike. Na srednjem nivoju, ki ločuje zgornjega od spodnjih podlijakov, se na vsaki izmed $\sqrt{K}$ robnih povezav nahaja medpomnilnik velikosti $K^{3/2}$. Ti medpomnilniki povezujejo zgornji lijak s spodnjimi tako, da se uporabijo kot podatkovni vhod za zgornji lijak in podatkovni izhod za spodnje lijake. To nam na vsakem nivoju da skupno velikost pomnilnikov K^2, ki se v podlija-

kih rekurzivno zmanjšuje. Medpomnilnike rekurzivno hranimo v pomnilniku znotraj lijaka, tako kot sama vozlišča drevesa. Slika 2.10 prikazuje sestavo lijaka.

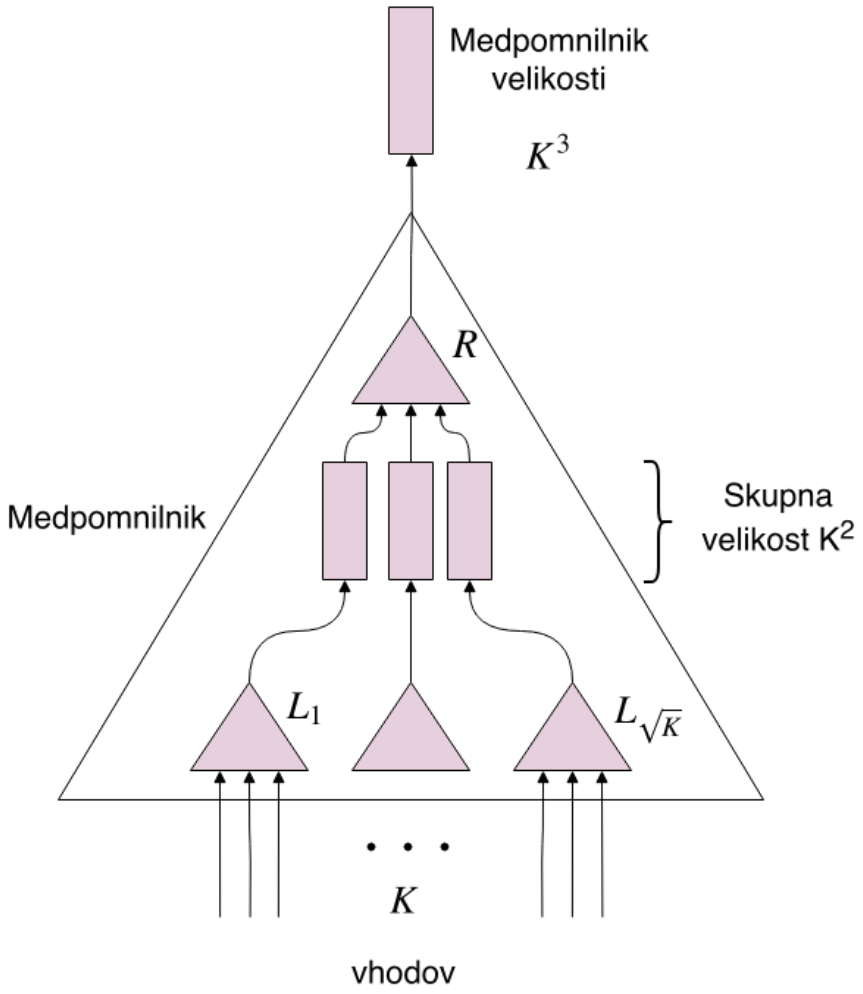
Zato, da lahko ostanemo konsistentni pri opisu algoritmov, bomo za izhod K-lijaka definirali medpomnilnik velikosti K^3. Tega medpomnilnika običajno ne hranimo v pomnilniku temveč si ga lahko predstavljamo kot izhodni mehanizem oziroma izhodno datoteko na zunanjem pomnilniku.

Težava v tej definiciji strukture je ta, da pogosto pride do težav pri določanju velikosti podlijakov in medpomnilnikov. V praksi ne moremo razdeliti K-lijak na $\sqrt{K}$-podlijake zato, ker K ni vedno kvadratno število. Standardni pristop za izogibanje težavam pri zaokroževanju bi bil, da bi za K vzeli najmanjše kvadratno število, ki zagotavlja, da so imajo vsi podlijaki za vrednost K kvadratno število. Na ta način bi dobili binarno drevo velikosti potence števila dve in bi posledično v najslabšem primeru dobili zelo prevelik lijak. Zato je v splošnem K-lijak sestavljen iz enega vrhnjega podlijaka ter iz j $\left\lceil\sqrt{K}\right\rceil$-lijakov in $\left(\left\lceil\sqrt{K}\right\rceil - j\right)$ $\left\lfloor\sqrt{K}\right\rfloor$-podlijakov kjer je $j \leq \sqrt{K}$ [13].

Urejevaje z uporabo lijaka deluje tako, da podatke najprej razdeli več manjših urejenih delov. Če imamo N vhodnih elementov, se podatke razdeli na $N^{1/3}$ urejenih delov velikosti $N^{2/3}$. Nato urjene dele zlije z uporabo K-lijaka, kjer je $K = N^{1/3}$.

Ob klicu K-lijaka skuša le-ta zapolniti svoj izhodni medpomnilnik velikosti K^3. To stori tako, da rekurzivno zahteva podatke iz svojih podlijakov. Obstajata dve različici algoritma imenovani dosledno (angl. eager) in zakasnjeno (angl. lazy) urejanje z lijakom.

Pri doslednem urejanju se polnjenje medpomnilnikov izvaja tako, da zgornji podlijak najprej preveri, če je vsak vhodni medpomnilnik poln. V primeru, da medpomnilnik ni poln izvede klic na spodnji podlijak, ki polni izbrani medpomnilnik. Polnjenje se izvaja dokler ne medpomnilnik vsebuje natančno $K^{3/2}$ elementov. V primeru, da podlijak ne more do konca napolniti medpomnilnika, ker mu je zmanjkalo vhodnih podatkov, le-ta zapiše elemente, ki so mu ostali in se označi kot iztrošen. Iztrošenega lijaka se na to



Slika 2.10: Grafični prikaz K-lijaka. K-lijak je rekurzivno sestavljen in $\sqrt{K}$ špodnjih" $\sqrt{K}$-lijakov $L_1, L_2, ..., L_{\sqrt{K}}$ in enega zgornjega $\sqrt{K}$-lijaka R.

v prihodnje ne kliče več. Psevdokoda za klicanje doslednega lijaka je prikaza v segmentu 2.1.

```
1  invoke(k-Funnel F)
2     if k je 2 ali 3 then
3        zlivaj podatke iz vhodov na izhod
4     else
5        repeat k3/2 krat do
6           for each buffer Bi, 0 <= i < k do
7              if Bi manj kot polovicno poln in Li ni iztrosen then
8                 invoke(Li)
9              fi
10          invoke(R)
11       od
12    fi
13    if ce smo na izhod postavili manj kot k3 elementov then
14       oznaci F kot izstrosen
15    fi
```

Algoritem 2.1: Psevdokoda za klicanje doslednega lijaka

Zakasnjeno urejanje z lijakom se od doslednega načina razlikuje po načinu polnjenja medpomnilnikov. Postopek polnjenja je poenostavljen tako, da se polnjenje medpomnilnika proži samo takrat, ko je medpomnilnik prazen. Na ta način ni več potrebno vsakič preverjati stanje medpomnilnikov in se odpravi odvisnost delovanja algoritma od velikosti medpomnilnikov. Psevdokoda za klicanje zakasnjenega lijaka je prikaza v segmentu 2.2.

```
1  lazy_fill(Vozliscce V)
2     while V-jev izhodni medpomnilnik ni poln do
3        if levi vhodni predpomnik prazen in levo hcerinsko vozlisce ni
             iztroseno then
4           lazy_fill(levo hcerinsko vozlisce)
5        if desni vhodni predpomnik prazen in desno hcerinsko vozlisce ni
             iztroseno then
6           lazy_fill(desno hcerinsko vozlisce)
7
8        if podatek na levem vhodu < podatek na desnem vhodu then
9           premakni podatek na levem vhodu na izhod
```

```
10      else
11         premakni podatek na desnem vhodu na izhod
12   od
13      if
14         levi vhodni predpomnik prazen in je levo hcerinsko vozlisce
               iztroseno in
15         desni vhodni predpomnik prazen in desno hcerinsko vozlisce ni
               iztroseno
16      then
17         oznaci V kot izstroseno
18      fi
```

Algoritem 2.2: Psevdokoda za klicanje zakasnjenega lijaka

Implementacija. Za eksperimentalno primerjavo smo implementirali zakasnjeno urejanje z uporabo lijaka. Algoritem najprej prešteje število podatkov v vhodni datoteki in določi vrednost K. Nato ustvari lijak in podatke porazdeli v poddatoteke. Sledi še povezovanje vhodov in izhodov lijaka z ustreznimi datotekami in klic rekurzivne procedure za polnjenje medpomnilnika glavnega lijaka.

Večinski del kode obsega gradnja lijaka. Naš algoritem lahko rekurzivno ustvari lijak poljubne višine tako, da pri gradnji podlijakov drevo razdeli na dva dela, kot je to opisano pri van Emde Boas porazdelitvi. Pri tem smo algoritem nekoliko poenostavili in zato omogoča le ustvarjanje polnega drevesa, saj je bilo za naše potrebe je bilo to dovolj. Z nekaj manjšimi spremembami pa je mogoče doseči, da se algoritem nekoliko bolje prilaga številu vhodnih podatkov. Algoritem za generiranje lijaka A.8.

Tako kot zahteva definicija so vozlišča in medpomnilniki rekurzivno shranjeni skupaj. To smo dosegli tako, da smo pomnilniški blok rezervirali v enem kosu in nato podatkovne strukture rekurzivno ustvarili.

Poglavje 3

Praktična primerjava

Algoritme za urejanje, ki smo jih opisali v teoretičnem poglavju smo tudi implementirali v programskem jeziku C. Za testiranje algoritmov uporabljamo naključno generirane binarne datoteke. Generator podatkov smo prenesli s spletne strani namenjene primerjavi zmogljivosti algoritmov za zunanje urejanje [14]. Z generatorjem smo ustvarili datoteko z 10000000 vnosi skupne velikosti približno 953 MB. Algoritme smo izvajali petkrat zapored in za primerjavo izbrali najboljše čase.

Algoritme smo izvajali na osebnem računalniku z naslednjo konfiguracijo:

	Opis
CPU	Intel Core 2 Quad Q6600
Predpomnilnik L1	4 x 32 KB
Predpomnilnik L2	2 x 4096 KB
RAM	2 x DDR2 2GB 400MHz
Trdi disk	Western Digital, 3600 RPM, 320 GB
Operacijski sistem	Lubuntu 15.10, 64bit

3.1 Osnovni algoritmi zlivanja

Pri tem poskusu smo med seboj primerjali osnovne algoritme, ki temeljijo na urejanju z zlivanjem. Želeli smo preizkusiti in prikazati vpliv izboljšav na število potrebnih branj in pisanj ter posledično tudi na hitrost urejanja. Med seboj smo primerjali navadno neuravnoteženo zlivanje, navadno uravnoteženo zlivanje in naravno uravnoteženo zlivanje.

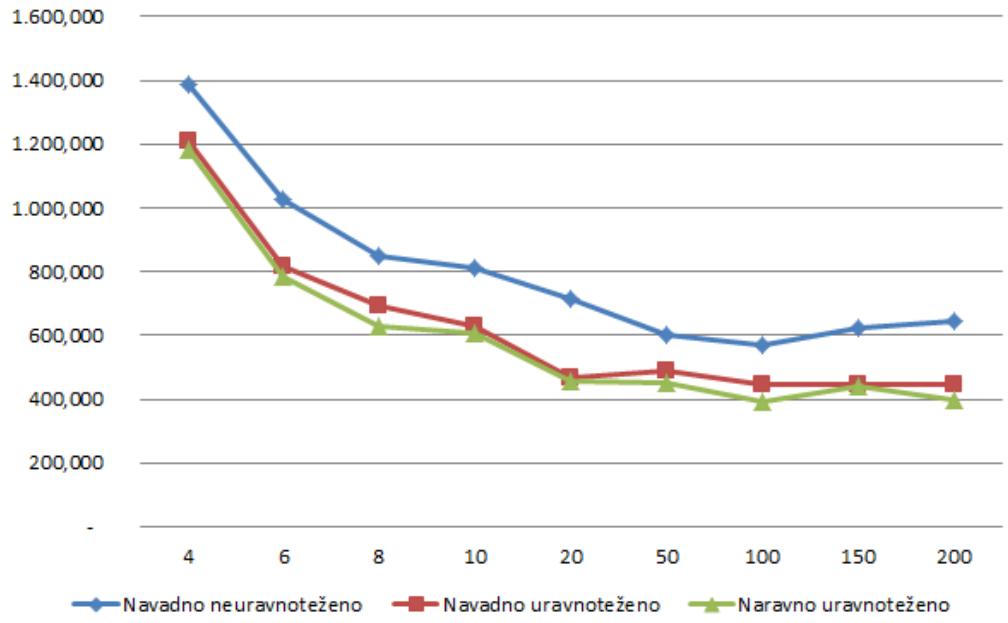
Kot pričakovano neuravnoteženo zlivanje zaradi dodatne faze porazdeljevanja porabi največ časa. Uporaba čet za urejanje ne pride do izraza, ker so podatki naključno urejeni.

Število trakov	4	6	8	10	20
Navadno neuravnoteženo	1389	1.027	850	810	714
Navadno uravnoteženo	1208	816	693	630	467
Naravno uravnoteženo	1183	783	628	606	458

Tabela 3.1: Prvi del rezultov. Časi izvajanja so v sekundah.

Število trakov	50	100	150	200
Navadno neuravnoteženo	602	571	623	645
Navadno uravnoteženo	486	445	444	446
Naravno uravnoteženo	451	392	440	396

Tabela 3.2: Drugi del rezultov. Časi izvajanja so v sekundah.



Slika 3.1: Na grafu je prikazan čas v sekundah, ki so ga za urejanje porabijo navadno neuravnoteženo in uravnoteženo ter naravno zlivanje. Največjo pospešitev doprinese odstranitev faze, ki zliva na en trak. Uporaba čet pa ne prispeva bistvene pospešitve.

3.2 Algoritmi za predurejanje

V tej primerjavi smo primerjali algoritme za predurejanje. Odločili smo se za primerjavo časa izvajanja in dolžine čet, ki jih algoritem ustvari. Čeprav opravijo algoritmi enako količino branj in pisanj, se časi izvajanja razlikujejo.

V tabeli 3.5 so podatki za navadno porazdeljevanje s četami, za datoteke z različnimi velikosti. Ker so podatki naključno razporejeni je povprečna dolžina čete zelo majhna. Predurejanje z notranjim urejanjem ustvari urejena zaporedja konstantnih dolžin, kot je prikazano v tabeli 3.6. Pri predurejanju s kopico pa smo eksperimentalno potrdili, da se ustvarijo čete približno dvakratne velikosti uporabljenega pomnilnika. Velikosti čet je prikazana v tabeli 3.7.

Število elementov	10000	20000	50000	100000
Celoten čas izvajanja	28,107	32,292	32,876	41,479
Čas izvajanja na CPU	12,503	13,945	15,428	18,359

Tabela 3.3: Predurejanje z notranjim urejanjem. Časi izvajanja so v sekundah.

Število elementov	10000	20000	50000	100000
Celoten čas izvajanja	30,201	29,718	30,459	39,903
Čas izvajanja na CPU	16,873	17,053	19,225	24,121

Tabela 3.4: Predurejanje s kopico. Časi izvajanja so v sekundah.

Število elementov	1000000	2000000	5000000	10000000
Število čet	500118	998883	2499920	5000809
Povprečna velikost čete	1,99	2,01	2,00	1,99

Tabela 3.5: Navadno porazdeljevanje s četami.

Število elementov	10000	20000	50000	100000
Število čet	1000	500	200	100
Povprečna velikost čete	10000	20000	50000	100000

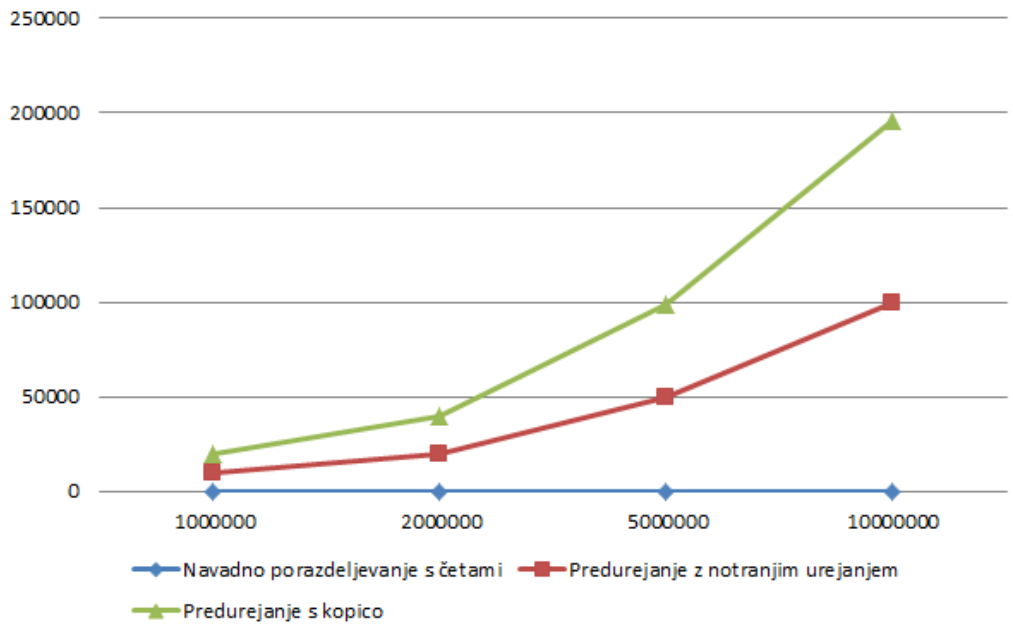
Tabela 3.6: Predurejanje z notranjim urejanjem.

Število elementov	10000	20000	50000	100000
Število čet	501	251	101	51
Povprečna velikost čete	19960,08	19960,08	99009,9	196078,4

Tabela 3.7: Predurejanje s kopico.

Ker smo želeli preveriti vpliv na algoritme za zunanje urejanje smo testirali izvajanje obeh algoritmov v kombinaciji s polifaznim in kaskadnim zlivanjem. Za testiranje smo uporabili dvajset trakov.

S številom enot pomnilnika označujemo, koliko elementov lahko algoritem



Slika 3.2: Na grafu je prikazana povprečna velikost čet, ki jih algoritmi sestavijo.

na enkrat hrani v pomnilniku.

Število enot pomnilnika	100000	200000	500000
Polifazno	197,4	168,8	109,3
Kaskadno	185,7	182,4	113,9

Tabela 3.8: Čas izvajanja pri predurejnaju s kopico. Časi izvajanja so v sekundah.

Število enot pomnilnika	100000	200000	500000
Polifazno	209,3	194,5	144,3
Kaskadno	205,5	189,0	161,4

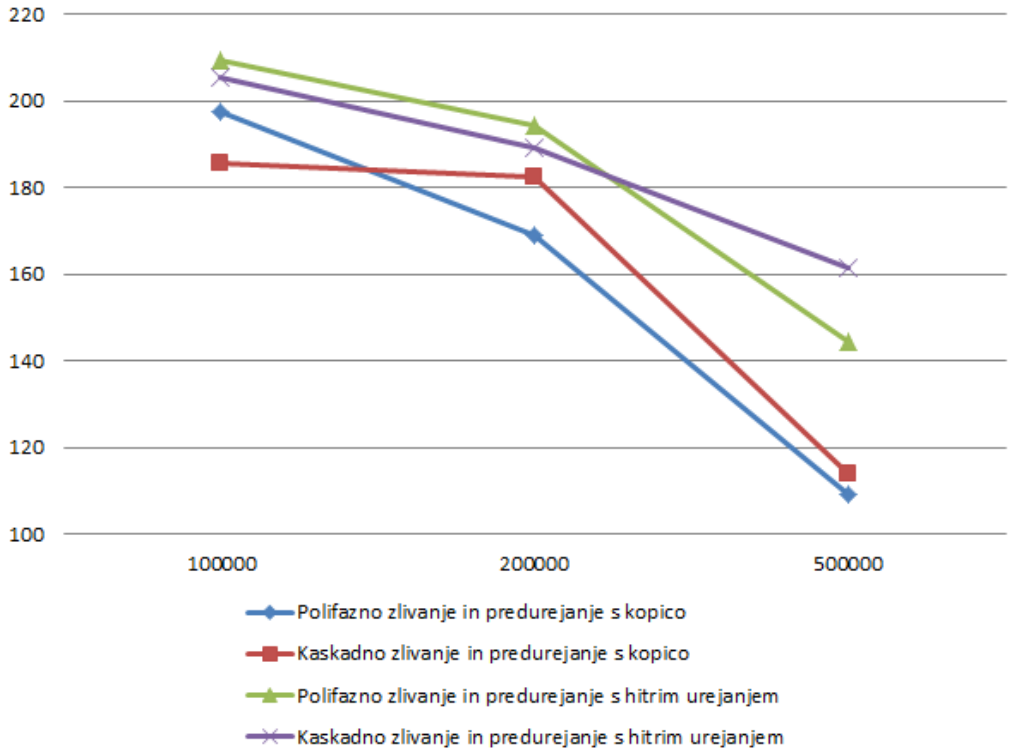
Tabela 3.9: Časi izvajanja pri predurejnaju s hitrim urejanjem. Časi izvajanja so v sekundah.

Število enot pomnilnika	100000	200000	500000
Polifazno	26765356	23131699	26765356
Kaskadno	30000000	30000000	20000000

Tabela 3.10: Število branj in pisanj pri predurejanju urejanju s kopico.

Število enot pomnilnika	100000	200000	500000
Polifazno	31400000	26600000	21000000
Kaskadno	40000000	30000000	30000000

Tabela 3.11: Število branj in pisanj pri predurejanju s hitrim urejanjem.



Slika 3.3: Slika prikazuje čase izvajanja naprednih algoritmov za urejanje glede pri uporabi predurejanja. Kot pričakovano so časi, kjer se uporablja predurejamje s kopico boljši.

3.3 Napredni algoritmi zlivanja

Polifazno in kaskadno zlivanje sta si zelo podobna. Medtem polifazno v vsaki fazi zlivanja ne obdela vseh podatkov, kaskadno pa manjše število faz zlivanja. Zato smo se odločili, da ju med seboj primerjamo. Ko se uporablja manj kot šetst trakov se polifazno zlivanje izede hitreje, pri večjem številu trakov pa ima kaskadno zlivanje prednost. Z večanjem števila trakov pada nivo porazdelitve hitreje pri kaskadnemu zlivanju kot pri polifaznemu. Manjši nivo porazdelitve pomeni, da se izvede manj faz zlivanja in posledično tudi manj prenosov podatkov.

Pri povečevanju števila trakov smo naleteli na še eno posebnost. S tem, ko povečujemo število trakov morata algoritma ob vsakem zlivanju izvesti vse več primerjav med elementi. Po določenem številu trakov postane čas, izvajanja na procesorju tako velik, da močno vpliva na celoten izvajalni čas.

Število trakov	4	6	8	10	20
Polifazno	818,7	666,5	624,6	638,6	602,4
Kaskadno	852,9	606,4	521,9	513,1	403,4

Tabela 3.12: Prvi del časov izvajanja algoritmov.

Število trakov	50	100	150	200
Polifazno	641,9	674,8	785,1	868,0
Kaskadno	431,6	412,5	450,4	491,2

Tabela 3.13: Drugi del časov izvajanja algoritmov.

Število trakov	4	6	8	10	20
Polifazno	146,9	118,2	110,2	121,1	120,8
Kaskadno	172,3	125,2	101,1	100,5	74,1

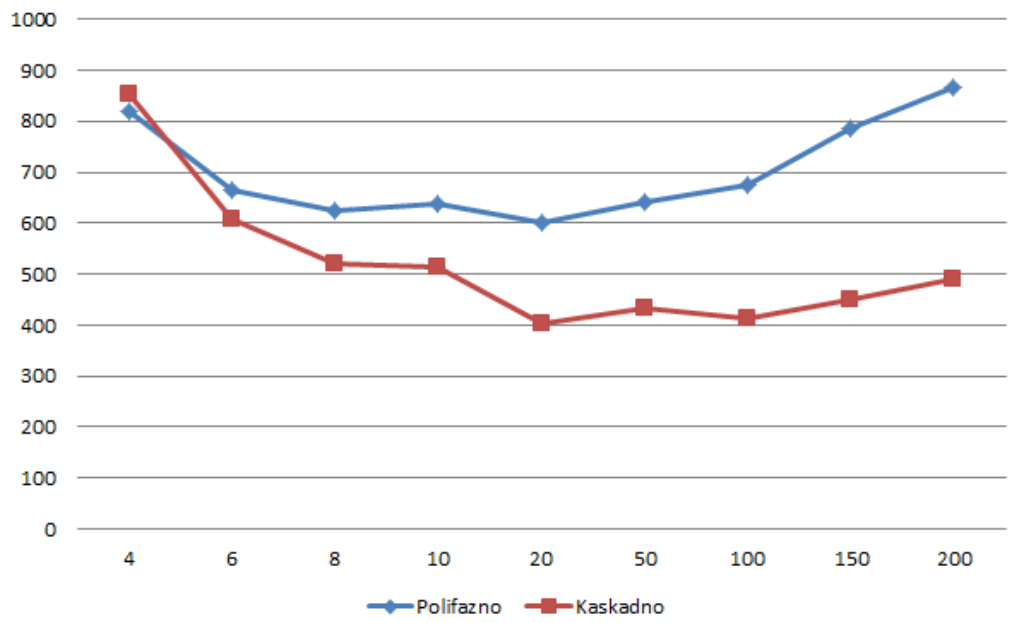
Tabela 3.14: Prvi del časov izvajanja algoritmov na CPU.

Število trakov	50	100	150	200
Polifazno	170,7	217,7	282,0	364,6
Kaskadno	86,0	94,2	116,2	142,5

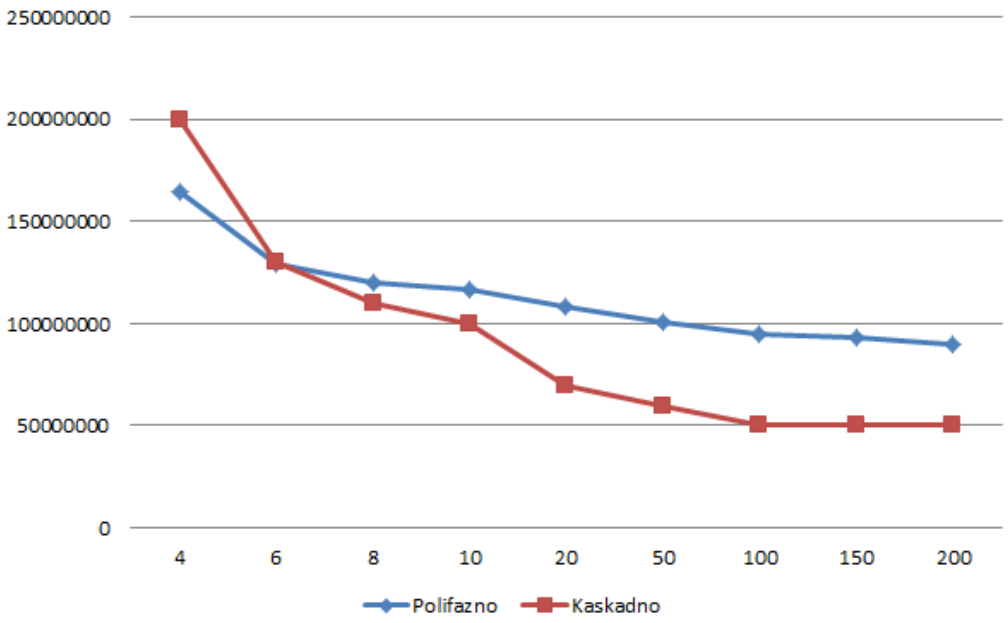
Tabela 3.15: Drugi del časov izvajanja algoritmov na CPU.

Število trakov	4	6	8	10	20	50	100	150	200
Polifazno	25	22	21	21	19	18	17	16	16
Kaskadno	19	12	10	9	6	5	4	4	4

Tabela 3.16: Nivo porazdelitve in število faz zlivanja.



Slika 3.4: Graf prikazuje čase izvajanja naprednih algoritmov za urejanje glede na izbrano število trakov.



Slika 3.5: Na grafu je prikazan število branj in pisanj, ki so jih opravili polifazno in kaskadno urejanje. Pri štirih trakovih opravi polifazno urejanje manjše število prenosov zato je nekoliko hitrejši.

3.4 Urejanje s porazdeljevanjem

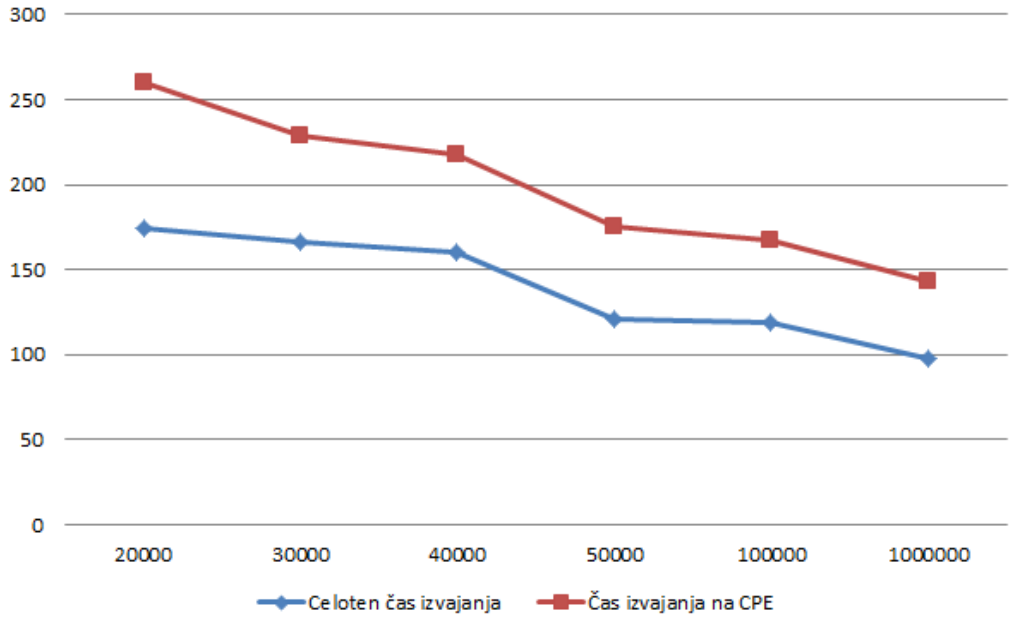
Pri tem preiskusu smo testirali hitrost algoritma za urejanje s porazdeljevanjem, v odvisnosti od uporabljenega pomnilnika. Pri večjem dodeljenem pomnilniku opravi manj rekurzivnih porazdelitev, a to pri našem testiranju ni prišlo do izraza. Število branj in pisanj se med testi naključno spreminja zaradi algoritma za naključno izbiranje porazdeljevalne množice. S spreminjanjem velikosti pomnilnika, se je spreminjal tudi procesorski čas. Ker algoritem lahko na enkrat v pomnilnik spravil večjo količino podatkov, je algoritem za notranje urejanje izvajanje pohitril.

Število elementov	20000	30000	40000
Celoten čas izvajanja	174,6	166,6	160,0
Čas izvajanja na CPU	85,0	61,7	57,4
Število branj	37593949	30987132	35641366
Število pisanj	24935713	24329481	25062935
Skupno število branj in pisanj	62529662	55316613	60704301

Tabela 3.17: Prvi del rezultatov izvajanja urejanja s porazdeljevanjem. Časi izvajanja so v sekundah.

Število elementov	50000	100000	1000000
Celoten čas izvajanja	120,8	118,7	97,2
Čas izvajanja na CPU	54,7	48,1	46,0
Število branj	37874065	37877185	33778836
Število pisanj	25032788	25212019	24005358
Skupno število branj in pisanj	62906853	63089204	57784194

Tabela 3.18: Drugi del rezultatov izvajanja urejanja s porazdeljevanjem. Časi izvajanja so v sekundah.



Slika 3.6: Graf, ki prikazuje čas izvajanja urejanja s porazdeljevanjem glede na število elementov, ki jih lahko na enkrat hrani v glavnem pomnilniku. Časi so v sekundah.

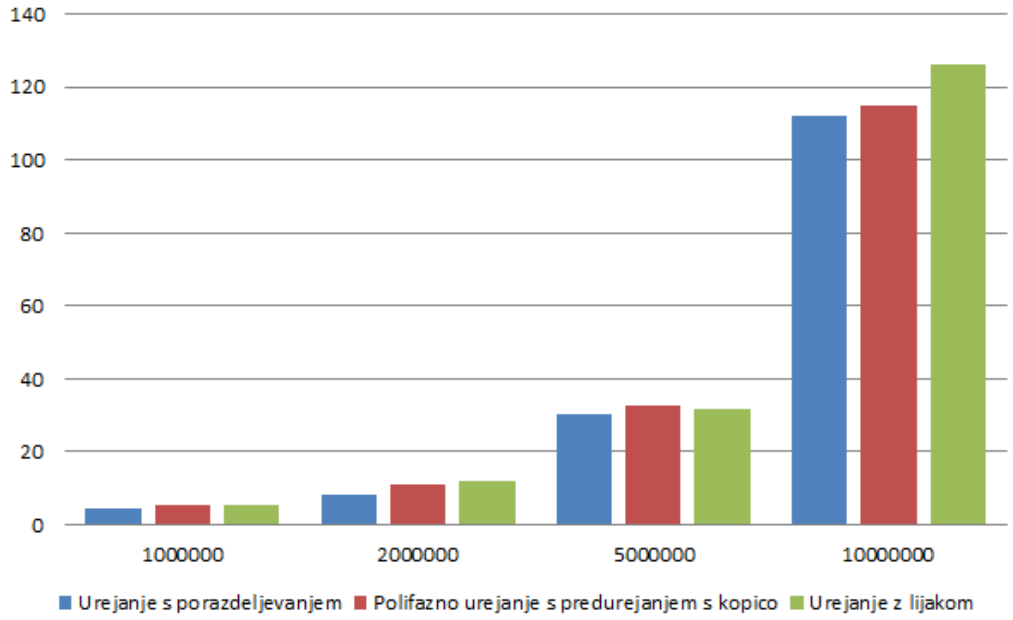
3.5 Kaskadno zlivanje, urejanje s porazdeljevanjem in zakasnjeno urejanje z lijakom

Ker urejanje z lijakom nima nobenih parametrov smo ga testirali tako, da smo primerjali izvajanje algoritma na različnih velikostih datotdone

ek. Pri tem smo izvajali najhitrejši algoritem za urejanje z zlivanjem in algoritem za urejanje s porazdeljevanjem. Pri kaskadnem urejanju smo uporabili dvajset trakov in predurejanje s kopico s pomnilniškim prostorom za 2000000 elementov. Urejanje s porazdeljevanjem je prav tako uporabljalo 2000000 elementov pomnilniškega prostora za porazdeljevanje.

Število elementov	1000000	2000000	5000000	10000000
Urejanje s porazdeljevanjem	4,411	8,418	30,361	112,2
Polifazno zlivanje	5,386	11,009	32,780	114,9
Urejanje z lijakom	5,583	12,113	31,685	126,1

Tabela 3.19: Rezultati izvajanja algoritmov nad datotekami z različnimi velikosti. Časi izvajanja so v sekundah.



Slika 3.7: Na grafu je prikazan čas v sekundah, ki so ga za urejanje različnih količin podatkov porabili algoritmi. Testirali smo urejanje s porazdeljevanjem, polifazno urejanje s predurejanjem z kopico in urejanje z lijakom. Algoritmi se kljub različnim velikostim datotek dobro obnesejo.

Poglavje 4

Sklepne ugotovitve

V diplomski nalogi smo predstavili področje zunanjega urejanja. Opisali smo osnove pomnilniške hierarhihje in omejitve, ki nam jih postavlja.

Predstavili smo več pristopov, ki skušajo omejitve zaobiti na različne načine. Od enostavnejših, ki so nastali v času, ko je bilo področje še mlado, do tistih naprednejših, ki skušajo čim bolje izkoristiti strojno opremo. Predstavili smo tudi napredno podatkovno sturkturo, na kateri temljijo relativno novi algoritmi, ki omogočajo optimalne čase izvajanja ne glede na strojno opremo na kateri se izvajajo in so kot taki primerni za vključevanje v programske knjižnice za splošno uporabo.

Na zadnje smo algoritme med sabo primerjali ter eksperimentalno potrdili prednosti in slabosti, ki smo jih opisali v teoretičnem delu. Kljub temu smo pokrili le majhen del področja in priložnoti za optimizacijo je še veliko.

Dodatek A

Izvorna koda v programskem jeziku C

Algoritem A.1: Zlivanje urejenih zaporedij na en trak

```
1  int merge(){
2     int i, cmp_index, all_empty, status;
3
4     for (i = 0; i < number_of_output_files; i++){
5        rewindFile(files_in, i);
6     }
7     files_out->file_pointer = fileOpen(files_out->file_name, "wb");
8
9     while (1){
10       all_empty = 1;
11       for (i = 0; i < number_of_output_files; i++){
12          if (!files_in[i]->end_of_file){
13             all_empty = 0;
14             files_in[i]->run_size = run_size;
15             files_in[i]->end_of_run = 0;
16          }
17       }
18       if (all_empty) break;
19
20       while (1){
```

```
21          cmp_index = getIndexOfLowestElement(files_in,
                number_of_output_files);
22          if (cmp_index == -1) break;
23
24          writeLine(files_out->file_pointer,
                files_in[cmp_index]->last_line);
25          status = readLine(files_in[cmp_index]->file_pointer,
                files_in[cmp_index]->last_line);
26
27          if (status <= 0){
28             files_in[cmp_index]->end_of_file = 1;
29          }
30          else{
31             files_in[cmp_index]->run_size -= 1;
32          }
33          if (files_in[cmp_index]->run_size == 0){
34             files_in[cmp_index]->end_of_run = 1;
35          }
36       }
37    }
```

Algoritem A.2: Spremenjena glavna zanka, ki zliva urejena zaporedja na več trakov

```
1
2          while (1){
3             cmp_index = getIndexOfLowestElement(files_in,
                  half_number_of_files);
4             if (cmp_index == -1)
5                break;
6
7             writeLine(files_out[fchoice_out %
                  half_number_of_files]->file_pointer,
                  files_in[cmp_index]->last_line);
8             status = readLine(files_in[cmp_index]->file_pointer,
                  files_in[cmp_index]->last_line);
9
10            if (status <= 0){
```

```
11              files_in[cmp_index]->end_of_run = 1;
12              files_in[cmp_index]->end_of_file = 1;
13            }
14            else{
15              files_in[cmp_index]->run_size -= 1;
16            }
17
18            if (files_in[cmp_index]->run_size == 0){
19              files_in[cmp_index]->end_of_run = 1;
20            }
21          }
22          fchoice_out++;
23        }
24  }
```

Algoritem A.3: Zlivanje s četami

```
1     do{
2       cmp_index = getIndexOfLowestElement(files_in,
            number_of_input_files);
3       if (cmp_index == -1)
4         break;
5
6       writeLine(files_out[fchoice_out %
            number_of_output_files]->file_pointer,
            files_in[cmp_index]->last_line);
7
8       strncpy(temp_line, files_in[cmp_index]->last_line, line_len);
9       status = readLine(files_in[cmp_index]->file_pointer,
            files_in[cmp_index]->last_line);
10
11      if (status > 0){
12        if (strcmp(temp_line, files_in[cmp_index]->last_line) > 0){
13          files_in[cmp_index]->end_of_run = 1;
14        }
15      }else{
16        files_in[cmp_index]->end_of_file = 1;
17        files_in[cmp_index]->end_of_run = 1;
```

```
18       }
19    } while (1);
```

Algoritem A.4: Polifazno zlivanje čet

```
1  void merge(){
2     int i, all_eof, all_dummies, cmp_index, temp_line[line_len], status;
3
4     fileStruct* file_tmp;
5     int file_last = number_of_files - 1;
6
7     for (i = 0; i < file_last; i++)
8        rewindFile(i);
9
10    while (fib_level){
11       while (1){
12          all_dummies = 1;
13          all_eof = 1;
14
15          for (i = 0; i < file_last; i++){
16             if (!files_array[i]->dummy_runs){
17                all_dummies = 0;
18                if (!files_array[i]->end_of_file){
19                   all_eof = 0;
20                   break;
21                }
22             }
23          }
24
25          if (!all_eof){
26             while (1){
27                cmp_index = getIndexOfLowestElement();
28                if (cmp_index == -1)
29                   break;
30
31                writeLine(files_array[file_last]->file_pointer,
                      files_array[cmp_index]->last_line);
32                strncpy(temp_line, files_array[cmp_index]->last_line,
```

```
                line_len);
33              status = readLine(files_array[cmp_index]->file_pointer,
                    files_array[cmp_index]->last_line);
34
35              if (status > 0){
36                 if (strcmp(temp_line, files_array[cmp_index]->last_line)
                       > 0){
37                    files_array[cmp_index]->end_of_run = 1;
38                 }
39              }
40              else{
41                 files_array[cmp_index]->end_of_file = 1;
42                 files_array[cmp_index]->end_of_run = 1;
43              }
44           }
45
46           for (i = 0; i < file_last; i++){
47              if (files_array[i]->dummy_runs)
48                 files_array[i]->dummy_runs--;
49              if (!files_array[i]->end_of_file)
50                 files_array[i]->end_of_run = 0;
51           }
52        }
53        else if (all_dummies){
54           for (i = 0; i < file_last; i++){
55              if (files_array[i]->dummy_runs)
56                 files_array[i]->dummy_runs--;
57           }
58           files_array[file_last]->dummy_runs++;
59        }
60
61        if (files_array[file_last - 1]->end_of_file &&
              !files_array[file_last - 1]->dummy_runs){
62           fib_level--;
63           if (!fib_level){
64              return;
65           }
66
```

```
67            fclose(files_array[file_last - 1]->file_pointer);
68            files_array[file_last - 1]->file_pointer =
                  fileOpen(files_array[file_last - 1]->file_name, "w+b");
69            files_array[file_last - 1]->end_of_file = 0;
70            files_array[file_last - 1]->end_of_run = 0;
71
72            rewindFile(file_last);
73
74            file_tmp = files_array[file_last];
75            memmove(files_array + 1, files_array, (number_of_files - 1) *
                  sizeof(fileStruct*));
76            files_array[0] = file_tmp;
77
78            for (i = 0; i < file_last; i++){
79               if (!files_array[i]->end_of_file)
80                  files_array[i]->end_of_run = 0;
81            }
82         }
83      }
84   }
85 }
```

Algoritem A.5: Kaskadno zlivanje

```
1  void merge(){
2     int i, j, all_dummies, cmp_index, temp_line[line_len], status,
          file_last;
3
4     fileStruct* file_tmp;
5     file_last = number_of_files - 1;
6
7     for (i = 0; i < file_last; i++)
8        rewindFile(i);
9     int merge_level = cascade_level;
10
11    for (; merge_level > 0; merge_level--){
12
13       for (file_last = number_of_files - 1; 1 < file_last; file_last--){
```

```
14          while (!files_array[file_last - 1]->end_of_file ||
                files_array[file_last - 1]->dummy_runs > 0){
15             all_dummies = 1;
16             for (i = 0; i < file_last; i++){
17                if (!files_array[i]->dummy_runs){
18                   all_dummies = 0;
19                   break;
20                }
21             }
22
23             if (all_dummies){
24                for (i = 0; i < file_last; i++){
25                   if (files_array[i]->dummy_runs)
26                      files_array[i]->dummy_runs--;
27                }
28                files_array[file_last]->dummy_runs++;
29             }
30             else{
31                while (1){
32                   cmp_index = getIndexOfLowestElement(file_last);
33                   if (cmp_index == -1)
34                      break;
35
36                   writeLine(files_array[file_last]->file_pointer,
                        files_array[cmp_index]->last_line);
37                   strncpy(temp_line, files_array[cmp_index]->last_line,
                        line_len);
38                   status = readLine(files_array[cmp_index]->file_pointer,
                        files_array[cmp_index]->last_line);
39
40                   if (status > 0){
41                      if (strcmp(temp_line,
                           files_array[cmp_index]->last_line) > 0){
42                         files_array[cmp_index]->end_of_run = 1;
43                      }
44                   }
45                   else{
46                      files_array[cmp_index]->end_of_file = 1;
```

```
47                   files_array[cmp_index]->end_of_run = 1;
48                }
49             }
50
51             for (i = 0; i < file_last; i++){
52                if (files_array[i]->dummy_runs)
53                   files_array[i]->dummy_runs--;
54                if (!files_array[i]->end_of_file)
55                   files_array[i]->end_of_run = 0;
56             }
57          }
58       }
59
60       rewindFile(file_last);
61
62       fclose(files_array[file_last - 1]->file_pointer);
63       files_array[file_last - 1]->file_pointer =
             fileOpen(files_array[file_last - 1]->file_name, "wb");
64    }
65
66    while (!files_array[0]->end_of_file){
67       writeLine(files_array[1]->file_pointer,
             files_array[0]->last_line);
68       status = readLine(files_array[0]->file_pointer,
             files_array[0]->last_line);
69       if (status <= 0)break;
70    }
71
72    rewindFile(1);
73
74    fclose(files_array[0]->file_pointer);
75    files_array[0]->file_pointer = fileOpen(files_array[0]->file_name,
          "wb");
76    i = 0;
77    j = number_of_files - 1;
78
79    while (i < j){
80       file_tmp = files_array[i];
```

```
81          files_array[i] = files_array[j];
82          files_array[j] = file_tmp;
83          i++;
84          j--;
85       }
86    }
```

Algoritem A.6: Algoritem za vstavljanje elementov kopico

```
1  status = readLine(file_input, temp_line);
2  if (status > 0){
3     if (strcmp(temp_line, heap[0]) >= 0){
4        swapStrings(&heap[0], &temp_line);
5        siftDownMin(heap, 0, size_temp - 1);
6     }
7     else{
8        swapStrings(&heap[0], &heap[size_temp - 1]);
9        swapStrings(&heap[size_temp - 1], &temp_line);
10       size_new++;
11       size_temp -= 1;
12
13       if (size_temp > 0){
14          siftDownMin(heap, 0, size_temp - 1);
15       }
16       else{
17          size_temp = size_new;
18          size_new = 0;
19          buildHeapMin(heap, size_temp);
20       }
21    }
22 }else{
23    heap++;
24    size_temp--;
25    quickSortAsc(heap, size_temp, line_len);
26    quickSortAsc(heap + size_temp, size_new, line_len);
27    size_temp = size_temp + size_new;
28    size_new = 0;
29 }
```

Algoritem A.7: Verjetnostno izbiranje porazdeljevalne množice elementov

```
1  void choseRandomPartitioningElements(char **dElementsPointer, fileStruct
       *file, int S){
2     int i;
3     S++;
4     int n = file->lines;
5     int d = max(1, rand_lim(n / (S + 1) - 1));
6     int dS = max(1, (d + 1) * (S - 1));
7     int randomDelay = rand_lim((n - dS) - 1);
8
9     char tmp[line_len];
10    file->file_pointer = fileOpen(file->file_name, "rb");
11
12    for (i = 0; i < randomDelay; i++){
13       readLine(file->file_pointer, tmp);
14    }
15
16    char *dsElements = (char*)malloc(dS * (line_len * sizeof(char)));
17    char **dsElementsPointer = (char**)malloc(dS * sizeof(char*));
18
19    for (i = 0; i < dS; i++)
20       dsElementsPointer[i] = dsElements + i * line_len;
21
22    readLines(file->file_pointer, dsElementsPointer, dS);
23
24    quickSortAsc(dsElementsPointer, dS, line_len);
25
26    for (i = 0; i < (S - 2); i++){
27       strncpy(dElementsPointer[i], dsElementsPointer[(i + 1)*(d - 1)],
             line_len);
28    }
29    fclose(file->file_pointer);
30    free(dsElementsPointer);
31    free(dsElements);
32 }
```

Algoritem A.8: Algoritem za rekurzivno generiranje K-lijaka

```
1  node* generateSubFunnel(int funnel_size, node
       **return_input_node_pointers){
2     int i, j;
3     node *output_node, *top_node, *bottom_top_node;
4
5     if (funnel_size == 2){
6        top_node = nodes_array + nodes_created;
7        return_input_node_pointers[0] = nodes_array + nodes_created;
8        top_node->num = nodes_created;
9        nodes_created++;
10       return top_node;
11    }
12
13    int k = funnel_size;
14    int height_total = log2((double)k);
15    int height_bottom = nextHighestPowerOf2(round((double)height_total /
          2));
16    int height_top = height_total - height_bottom;
17
18    int size_of_top_subfunnel = pow((double)2, height_top);
19    int size_of_bot_subfunnels = pow((double)2, height_bottom);
20
21    int mid_buffers_size = round((double)root(pow((double)k, 3), 2));
22
23    node **top_input_nodes = (node**)malloc((size_of_top_subfunnel / 2)*
          sizeof(node*));
24    node **bottom_input_nodes = (node**)malloc((funnel_size / 2) *
          sizeof(node*));
25
26    /* Generiraj prvo - vrhnje pod-drevo ( A ) */
27    for (i = 0; i < (size_of_top_subfunnel / 2); i++){
28       top_input_nodes[i]->left_input_buffer->buffer_pointer =
             buffers_array_pointer;
29       top_input_nodes[i]->left_input_buffer->buffer_size =
             mid_buffers_size;
30       buffers_array_pointer += mid_buffers_size * (line_len *
             sizeof(char));
31
```

```
32        top_input_nodes[i]->right_input_buffer->buffer_pointer =
              buffers_array_pointer;
33        top_input_nodes[i]->right_input_buffer->buffer_size =
              mid_buffers_size;
34        buffers_array_pointer += mid_buffers_size * (line_len *
              sizeof(char));
35     }
36
37     //Spodnji del - ( B1 ... Bn )
38     for (i = 0, j = 0; i < size_of_top_subfunnel; i += 2, j++){
39        bottom_top_node = generateSubFunnel(size_of_bot_subfunnels,
              bottom_input_nodes + i*(size_of_bot_subfunnels / 2));
40        top_input_nodes[j]->left_node = bottom_top_node;
41        bottom_top_node->output_buffer =
              top_input_nodes[j]->left_input_buffer;
42
43        bottom_top_node = generateSubFunnel(size_of_bot_subfunnels,
              bottom_input_nodes + (i + 1)*(size_of_bot_subfunnels / 2));
44        top_input_nodes[j]->right_node = bottom_top_node;
45        bottom_top_node->output_buffer =
              top_input_nodes[j]->right_input_buffer;
46     }
47
48     for (i = 0; i < funnel_size / 2; i++){
49        return_input_node_pointers[i] = bottom_input_nodes[i];
50     }
51
52     free(top_input_nodes);
53     free(bottom_input_nodes);
54     return output_node;
55  }
```

Literatura

[1] J. S. Vitter, "External memory algorithms and data structures: Dealing with massive data," *ACM Comput. Surv.*, vol. 33, pp. 209–271, June 2001.

[2] B. Gregg, *Systems Performance: Enterprise and the Cloud*. Upper Saddle River, NJ, USA: Prentice Hall Press, 1st ed., 2013.

[3] J. S. Vitter, *Algorithms and Data Structures for External Memory*. Hanover, MA, USA: Now Publishers Inc., 2008.

[4] B. Vilfan, *Osnovni algoritmi*. Fakulteta za računalništvo in informatiko, 1998.

[5] W. Lynch, "The f-flbonaccl numbers and polyphase sorting,"

[6] N. Wirth, "Algorithms and data structures," 1986.

[7] R. Gilstad, "Polyphase merge sorting: an advanced technique," in *Papers presented at the December 13-15, 1960, eastern joint IRE-AIEE-ACM computer conference*, pp. 143–148, ACM, 1960.

[8] D. E. Knuth, *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998.

[9] E. D. Demaine, "Cache-oblivious algorithms and data structures," *Lecture Notes from the EEF Summer School on Massive Data Sets*, vol. 8, no. 4, pp. 1–249, 2002.

[10] H. Wang and B. Lin, "Pipelined van emde boas tree: Algorithms, analysis, and applications," in *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*, pp. 2471–2475, IEEE, 2007.

[11] H. Prokop, *Cache-oblivious algorithms.* PhD thesis, Massachusetts Institute of Technology, 1999.

[12] G. S. Brodal, R. Fagerberg, and K. Vinther, "Engineering a cache-oblivious sorting algorithm," *J. Exp. Algorithmics*, vol. 12, pp. 2.2:1–2.2:23, June 2008.

[13] K. Vinther, *Engineering cache-oblivious sorting algorithms.* PhD thesis, Aarhus Universitet, Datalogisk Institut, 2003.

[14] "Sort Benchmark Home Page." `http://sortbenchmark.org/`. Zadnji dostop dne 1.9.2015.